INTEGRATED COMPUTATIONAL AND NETWORK QOS IN GRID COMPUTING

GOKUL PODUVAL

(B.Eng.(Computer Engineering), NUS)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

NATIONAL UNIVERSITY OF SINGAPORE

2005

Acknowledgments

I would like to express my gratitude to my supervisor, Dr. Tham Chen Khong for his guidance and support throughout my research. I am grateful to my colleagues in the Computer Networks and Distributed Systems Lab (CNDS), National University of Singapore (NUS), especially Daniel Yagan (CNDS) who provided me his implementation of the SMART algorithm, and Yeow Wai Leong (CNDS) for helping me with the grid setup.

I am indebted to Anthony Sulistio and Dr. Rajkumar Buyya from GRIDS Lab, University of Melbourne for helping me to integrate my work with GridSim.

I am thankful to NUS for providing me financial support for my research.

This dissertation is dedicated to my parents and my sister, for their encouragement and support at all times.

Contents

List of Tables

List of Figures

List of Symbols

$\mathbb{S}$	Set of States
$\mathbb{A}$	Set of Actions
s	State
a	Action
T	State Transition Function
R,r	Reward
$\mathcal{AR}$	Average Reward
Q	Q-Value
V	Value function
α,β	Learning Rate
γ	Discount Rate
ϵ	Exploration Rate
λ	Trace-decay parameter
δ	Temporal Difference Error
π	Policy
π^*	Optimal Policy
e	Eligibility Trace
τ,t	Time
θ	Temperature
μ	Bandwidth
ρ	Response Time

ϕ	Weight Assigned to a Class or Flow
ς	Scaling Factor
Γ	Packet Size
Λ	Arrival Time of a Packet
C	Capacity of a Server
D	Delay Bound
F	Finish Time
E	Expected Value
$P(a)$	Probability of 'a'
TS	Amount of Service Received
c	Class
v	Virtual Clock

List of Abbreviations

AD	Administrative Domain
AF	Assured Forwarding
API	Application Programming Interface
CBQ	Class Based Queuing
CMAC	Cerebellar Model Articulate Controller
CPU	Central Processing Unit
DCM	Darken-Cheng-Moody
DiffServ	Differentiated Services
DP	Dynamic Programming
FIFO	First In First Out
FQ	Fair Queuing
FRED	Flow-based Random Early Detect
GIS	Grid Information Server
GPS	General Processor Sharing
GR	Grid Resource
GRACE	Grid Architecture for Computational Economy
GS	Guaranteed Service
GSP	Grid Service Provider
ICMP	Internet Control Message Protocol
IP	Internet Protocol
IntServ	Integrated Services

I/O	Input/Output
MIPS	Million Instructions Per Second
MDP	Markov Decision Process
MLP	Multi-Layer Perceptron
MTU	Maximum Transmission Unit
NDP	Neuro-Dynamic Programming
NMS	Network Management System
NWS	Network Weather Service
OSPF	Open Shortest Path First
P2P	Peer-to-Peer
PHB	Per Hop Behavior
PQ	Priority Queuing
QoS	Quality of Service
RIP	Routing Information Protocol
RBF	Radial Basis Function
RL	Reinforcement Learning
RSVP	Resource reSerVation Protocol
RTT	Round Trip Time
SCFQ	Self Clocked Fair Queuing
SLA	Service Level Agreement
SMART	Semi-Markovian Average Reward Technique
SMDP	Semi-Markov Decision Process
SNMP	Simple Network Management Protocol
TCP	Transmission Control Protocol
TD	Temporal Difference
ToS	Type of Service
UB	User Broker
UDP	User Datagram Protocol

UML	Unified Modeling Language
VO	Virtual Organization
WFQ	Weighted Fair Queuing

Summary

Grid computing is a technology that allows organizations to lower their computing costs by allowing them to share computing resources, software licenses and storage media. As more services are pushed to grid networks in the future, Quality of Service (QoS) will become an important aspect of this service.

In this thesis, we look into a method for providing QoS through learning and autonomic methods. The learning methodology we use is known as Reinforcement Learning (RL), a stochastic optimization method used in areas like robotics. An autonomous method is one in which no manual intervention is required, and a major aim in this thesis is to provide QoS in such a manner. RL based systems will help achieve this, since they are model free, and require no supervision to learn. An autonomous system will not require constant monitoring, and if well designed, will be able to maximize the utility of the grid.

We explore two RL methods, known as Watkins' Q(λ) and Semi-Markovian Average Reward Technique (SMART), to perform resource allocation on computing and network resources. We also explore two alternatives to resource allocation, provisioning and reservation. We performed simulations by selectively enabling our proposed solution on user's grid brokers and agents located at networking and computing resources. We have evaluated the performance of our learning methodology in simulation using a grid simulation software known as GridSim. Since earlier versions of GridSim did not support networking resources like routers and network links, we extended GridSim to provide this functionality. The design of this network functionality is covered in brief. We also performed experiments on a testbed in order to support the observations made from the analysis of our simulations. This thesis describes in detail the design of our proposed solution to providing QoS, the experiments performed to verify our solution the conditions under which the experiments were carried out, and what we can infere from the

results.

From the simulation and implementation results, we conclude that reinforcement learning methods are able to adapt successfully to a given scenario. Resource allocation techniques relying on RL methods are able to modify their allocation levels to support the workload provided to the system. These methods work better than Static methods of resource allocation. We also conclude that reservation of resources can provide better a quality of servicve then provisioning methods.

Keywords: Grid Computing, Reinforcement Learning, Watkins Q(λ), SMART, GridSim

List of Publications

- A. Sulistio, G. Poduval, R. Buyya and C.-K. Tham, "On Incorporating Differentiated Network Service into GridSim", submitted for approval to *Grid Technology and Application*, a special section with *Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods, and Applications*, Elsevier Publications , 2006.

- C.-K. Tham, and Gokul Poduval, "Adaptive Self-Optimizing Resource Management for the Grid", to appear in *The 3rd International Workshop on Grid Economics and Business Models (GECON '06)*, Singapore, 2006

- A. Sulistio, G. Poduval, R. Buyya and C.-K. Tham, "Constructing a Grid Simulation with Differentiated Network Service using GridSim", in *Proceedings of The 2005 International MultiConference in Computer Science & Computer Engineering (ICOMP '05)*, Las Vegas, Nevada, USA, 2005.

- G. Poduval and C.-K. Tham, "A Neuro-Dynamic Approach to Resource Provisioning in Computational Networks", in *The 2004 Australian Telecommunication Networks and Applications Conference (ATNAC '04)*, Sydney, Australia, 2004.

Chapter 1

Introduction

Grid Computing has emerged as a powerful way to maximize the value of computing resources. Grid computing allows organizations and people to unite their computing, storage and network systems into a virtual system which appears as one point of service to a user. Grid computing can be something as simple as a collection of similar machines at one location to a mix of diverse systems spread all over the world. The machines may not even be owned by a single entity. Organizations can link to other organizations computing resources for collaborative problem solving in various fields of science, engineering, medicine etc. These resources are shared with strict rules and are highly controlled, with resource providers and consumers defining clearly what is being shared, the conditions of them being shared, cost, time of use, specific use etc. A set of individuals and/or institutions sharing resources under such conditions is known as a *Virtual Organization* (VO). [1].

Some of the most primitive forms of Grid computing were projects like SETI@Home [2] and Distributed.net [3]. These projects harnessed the spare computing cycles of PCs distributed all around the world for a single purpose. These applications are also known as *Peer-to-Peer (P2P)* networks.

1.1 Services Provided by Grid Computing

Grid computing can provide organizations the following features [4].

- Exploit underutilized resources - Many computers in organizations are only active at certain times of the day. For e.g., desktop machines used by people are only used during office hours. The computing cycles of these machines can be harnessed with the use of grid computing to run compute intensive jobs. This is especially easy if the process to be executed is easily parallelized. This process is also known as *scavenging*.

- Enable Virtual Resources and Virtual Organizations - Grid computing simplifies collaboration between different organizations . Different users can be divided into different virtual organizations , with each VO having a different policy for sharing of resources. Storage space, data, application licenses and computing power etc. can be shared by people who are in different physical organizations, but in the same VO. They can have common security policies, and can implement payment systems for the usage of resources. To a single user, all resources within a VO can be consolidated to feel like a single resources.

- Provide Resource Balancing - In a grid, a large number of machines are federated into a single entity. If there is a sudden spike in the demand for a computing resource, a good scheduling policy can split the request into various resources that form the grid.

- Establish A Grid Computing Market - The fact that users from different organizations can use resources from other organizations transparently is giving rise to services like computing and storage farms. Entities known as the Grid Service Provides can provide users access to large computing resources, application libraries, network bandwidth and storage at specified

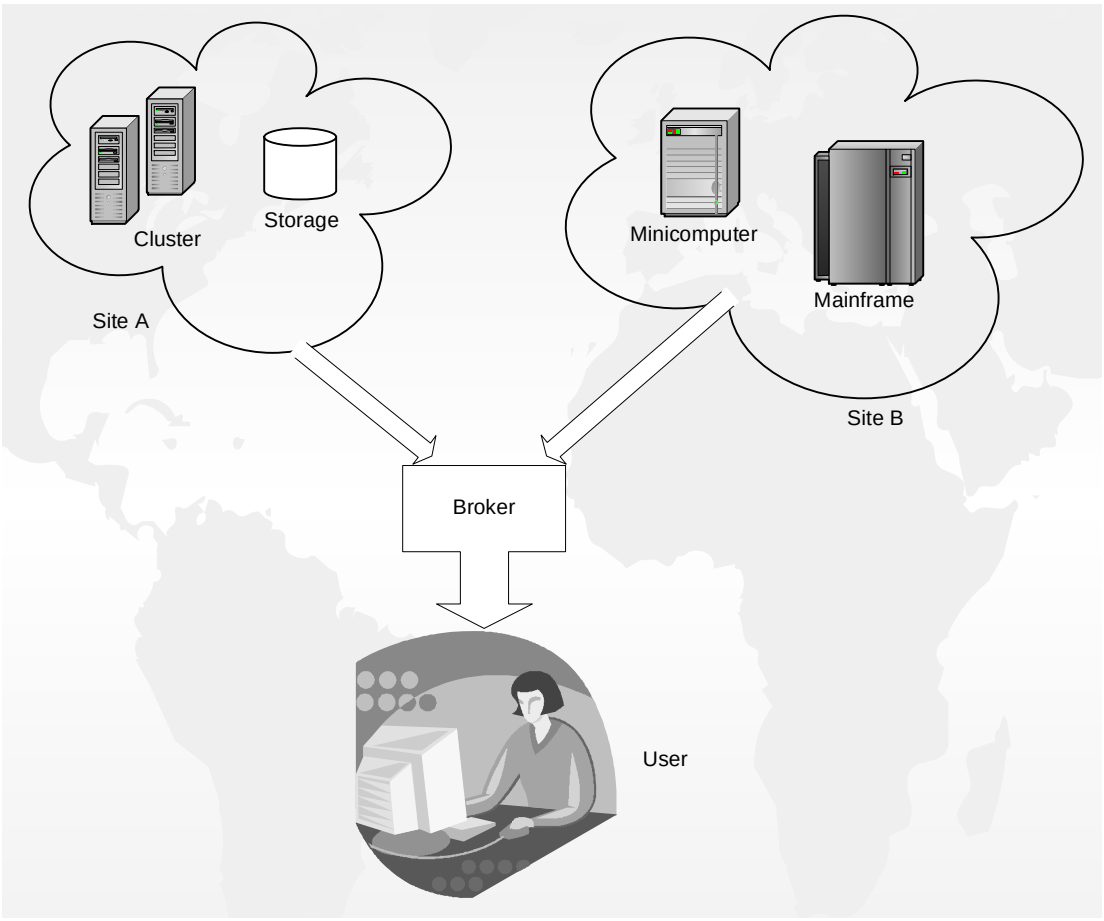

Figure 1.1: A Virtual Organization Aggregates Resources in Various Domains to Appear as a Single Resource to the End-user

rates. This is convenient for the users too since it lowers the cost for occasional use of such resources. Organizations with large computing resources can recoup their investments by lending out their idle computing prowess. This model is also referred to as *Utility Computing*.

- Provide Quality of Service - In various organizations, some projects could be of higher importance than the others due to its monetary value or because of time constraints. Users could implement brokers, who can negotiate on their behalf with grid service providers to lease more computing power, storage space, network bandwidth etc. In a utility computing model, service providers can charge premium rates for providing expedient service.

1.2 Need for Job Classes

Jobs that run of Grid Networks may be of several types. Some jobs may require large amounts of computational power, while others may require quick response times. From the point of view of service providers, it is not be possible to specify what kind of service each job should receive. Better service can be provided if jobs are profiled, and split into separate classes. Each job in the same class receives the same service from the grid network and network providers.

In this thesis, jobs are divided into classes depending on the amount of processing power they require, the amount of data to be transferred, and its deadline. Jobs belonging to the same class have similar values for each of these parameters.

1.3 Quality of Service

Quality of Service refers to the ability of a node to provide differential service to different classes of jobs. In order to do this, the resource providing QoS needs to be aware of incoming workload to be of different classes, and it should have inbuilt

mechanisms to provide varying share of its capability, depending on a specified policy. In our thesis, we have analyzed QoS requirements for two kinds of resources - Processing nodes and Network elements.

1.3.1 QoS for Processing Nodes

[5] discusses some of the metrics relevant to Quality of Service in Grid Computing. The QoS of a grid job at a processing node can be defined in terms of the following parameters -

- *Latency* of a task is the amount of time taken by the grid node to execute it. It can also be said to be the response time of the node. Latency is the most important parameter for jobs that need immediate attention, for e.g. a system that keeps tabs on the stock market to decide which stocks to invest in.

- *Throughput* is the units of work that is accompanied by the grid service in unit time. The throughput for a certain job can be measured as the amount of work provided by the job divided by the amount of time taken to complete it i.e. latency. Throughput is an important factor for jobs that need lot of processing power for e.g. weather simulations, nuclear reaction simulations or processing large financial worksheets.

- *Availability* - Another important QoS metric is the availability. Availability is defined as the fraction of time that the resource is available for use. For e.g., the desktop PCs may only spare their CPU cycles for grid services at night. However, some jobs may require guarantees of 100% availability of service, for e.g. a fire or defense alarm system. Systems that require higher availability may need more dedicated hardware.

1.3.2 QoS for Network Elements

In networks, QoS refers to the capability of a network to provide differentiated service to different types of network traffic. Certain flows should be able to obtain better service than other flows. Common methods to do this include providing higher bandwidth to high priority flows and/or dropping more packets of low priority packets in the presence of congestion. In computer networks, some of the important QoS parameters are bandwidth, packet loss rate, delay and jitter.

- *Bandwidth* - Bandwidth refers to the amount of data that can be sent in unit time by a flow. Applications which require lot of I/O to and from network resources can be sped up by providing them higher bandwidth.

- *Packet Loss Rate* - In IP networks, packets are not guaranteed to reach their destination. If queues at any intermediate router in a flow are full, an incoming packet is dropped. Flows that are guaranteed low packet loss rates can be accommodated by dropping packets of lower priority, when high priority packets arrive at a router that has full queues. Flow-based Random Early Detect (FRED) ([6]) is a common technique that can be used to achieve preferential treatment with regard to loss rate for packets of certain classes.

- *Delay* - In computer networks, delay is composed of transmission delay, prorogation delay and queuing delay. Transmission delay depends on the bandwidth provided to the flow and prorogation delay depends on the speed of electrons in metallic mediums, or speed of light in optic fibers. Queuing delays occur at a router because routers have an overhead for processing each packet, and many packets in its queues that delay the time at which a specific packet will be processed by the router. Routers which guarantee low delays to certain flows can use buffer management techniques like Priority Queuing (PQ), Weighted Fair Queuing (WFQ) or Class Based Queuing (CBQ).

- *Jitter* - Many applications are sensitive to the variation in delay, rather than the actual delay itself. This is usually the case with applications that rely on a regular arrival of data, for e.g. a video client. Variation in delay of packets is known as jitter. Jitter can be kept low by using small queue sizes in routers.

[7] provides a detailed view of QoS in networks and the various mechanisms used to implement and provide QoS in networks..

1.3.3 QoS Levels

Quality of Service (QoS) can be distinguished into two categories -

- *Soft QoS* - Soft QoS consists of providing better service to some jobs on resources like network and processing nodes. Service Providers can make some assurances about bandwidth, delay, processing power etc., by treating some jobs better than the rest. However, no guarantees are made, and the assurances are given on a statistical basis. Soft QoS is mainly achieved by having multiple classes of service, with different priorities. One example of a soft QoS mechanism is the Assured Forwarding Per-Hop-Behavior (PHB) ([8] of DiffServ ([9]).

- *Hard QoS* - In some cases, soft QoS may not be enough to satisfy all the requirements of an user. For jobs with tight deadlines, the customer may require the service provider to guarantee that the job will get completed within a certain time. Hard QoS includes mechanisms for guaranteeing the bandwidth or other QoS metrics at resources. Hard QoS allows the resource provider to decouple the requirement of one customer from the load introduced by another, since he needs to make sure there are enough resources available to satisfy both. Though this service requires more stringent resource management techniques, as compared to soft QoS, service providers

may be interested in providing such services since they can charge a premium rate for it. An example of Hard QoS is the Resource ReSerVation Protocol (RSVP) ([10]). RSVP can be used by an application to make resource reservation at each node that the application stream will be traversing. Another example is the Guaranteed Service (GS) ([11]) provided by the IntServ ([12]) framework.

1.4 Provisioning and Reservation

We discussed in Section 1.3.3 the difference between soft and hard QoS. Resources that support soft QoS are said to support provisioning. In provisioning, the service levels supported are only statistical. Resources provided differentiated service to multiple classes of jobs. Load from other sources can affect the performance of a job in provisioning schemes. Reservation, on the other hand, refers to hard QoS guarantees provided by resources. Here, the service parameters are absolute, and do not change depending on the load at a resource.

1.5 Service Level Agreements

Users and service providers can agree to the service parameters that the user will receive from a service provider through a Service Level Agreement (SLA). A SLA contains the contract between the subscriber and the service provider regarding the characteristics of traffic allowable at the service providers ingress node. For example, a SLA could state that a subscriber will always receive a lower delay for his packets as compared to other flows, as long as the number of packets are kept below a certain threshold. This is an example of soft QoS. A SLA could also state that a job of certain size will always get processed within an hour at a processing node. This is an example of hard (guaranteed) QoS. [13] provides examples of how static of dynamic SLAs can be used in conjunction with other technologies

like IntServ and DiffServ to provide QoS to users.

The major problem with SLAs is that it leads to wastage of resources at the service providers side. If the mechanisms ensuring QoS are static, the service provider needs to make sure that he has enough capacity to cater to every user consuming their maximum allowable capacity. This leads to underutilization of his resources when the subscribers are sending below their upper limits. Thus, static QoS mechanisms do not provide maximum value for money.

1.6 Integrated Network and Processing QoS

In grid computing models, users submit their jobs via brokers to one or many grid processing nodes. These processing nodes may be within the same network domain, or could be a few network Administrative Domains (AD) away. If a grid user wants his jobs to meet certain QoS criteria, he needs the following

i. The grid service provider must have mechanisms to ensure that he can provide QoS to the jobs, for e.g. higher or fixed share of processing capacity, or ensuring 24/7 availability.

ii. The network domains through which the jobs passes also need to provide QoS support. This can be through means of providing priority queuing, higher share of bandwidth, or allowing packets to queue at the head of the router queues.

Jobs can only fulfill their deadline and other QoS requirements if QoS support is provided by both of the above. For e.g. if only grid nodes provide QoS support, data packets for that job may get held up in the network, or even lost, leading to QoS violations overall for the job. This will be especially true if the job is I/O intensive.

One way in which the user can make sure of his job meeting deadlines, is to sign a SLA(1.5) with the grid service provider, and all the network domains that

his data will be passing through. However, this would mean the location of the grid processing nodes will need to be fixed and known in advance. This takes away a lot of flexibility of having a grid system that is supposed to be transparent to the user. Also, in services like utility computing, the user sends his jobs to the utility provider, from where it may be farmed out to any location the provider thinks is appropriate. Thus, signing SLAs in advance is a cumbersome process which requires manual intervention, and does not always fit in well with the grid and utility computing model.

In this thesis, we have used a model where network and processing resources always provide QoS support. The services a job receives depends on its class. Class 1 jobs receive preference over Class 2 jobs. The resource allocations among classes are decided by policies at each resource. The policies are learnt and continually adjusted by agents residing at each resource.

1.7 Related Work

Providing Quality of Service in Grids is a challenging problem and many researchers have taken a look at it in recent years. [14] presents a resource management architecture called GARA, that addresses the problem of achieving end-to-end QoS guarantees across heterogeneous collections of shared resources. It works in conjunction with the Globus Toolkit. ([15]). It allows the construction of reusable co-reservation and co-allocation agents that can combine domain and resource specific knowledge to discover, reserve and and allocate resources to try and meet application QoS requirements. [16] discusses the principles behind the GARA architecture. However, GARA requires that the application use an API to create calls for reservation and allocation.Also, the project seems to be out-of-date and doesn't work with newer versions of Globus, with the last update to their website in June 2000.

[17] proposes a Grid Architecture for Computational Economy (GRACE). GRACE is an economic framework for consumers and service providers to operate in a grid computing market. The consumers interact with brokers to express their budget and deadline requirements. The resource broker is responsible for discovering grid resources, negotiating with GSPs and controlling and scheduling jobs. GRACE proposes setting up a market directory, where resource owners can publish their services along with their service parameters like pricing policies. It also proposes setting up a Grid Bank which records resource usage, bills consumers, and transfers funds to service providers. The pricing strategies can be based on flat rates, demand-and-supply, calendar based, bargaining based etc. [17] also discusses Nimrod-G, which is a tool for automated modelling and execution of applications, and it is based on the GRACE framework. The GRACE framework enables QoS by allowing users and service providers to negotiate the price and deadline of job. However, it does not propose any mechanisms by which the service providers can ensure that they deliver the QoS they promised when publishing their services in the market directory. It also does not explore how resource providers can maximize the usage of their grids.

[18] details another approach to provide QoS to grid services using a framework called QoSINUS. This approach aims to provide an end-to-end Best Effort QoS. Only network level QoS is provided in this approach. Programs can specify their QoS parameters through an API provided by QoSINUS, and the QoSINUS service tries to map their requests with a class of IP service on a network that supports some form of QoS like DiffServ ([9]). The QoSINUS approach contains adaptive control components that are responsible for class mapping and adapting the packet marking policy according to the performance experienced by the packets of each particular flow. The adaptive policies can be flexible, ranging from static mapping to policies that give higher priorities to jobs with earlier deadlines. There is no provision for CPU Reservation in this framework, therefore no guarantees can be

provided. The QoS provided is very basic, limited to Best Effort Service. Though this framework has adaptive algorithms, the adaptive algorithms are simple in nature.

[19] proposes a framework called *Grid QoS Management (G-QoSM)*, which is compatible with the Open Grid Services Architecture (OGSA) specification. OGSA is a framework to build Virtual Organizations, and defines grid services in the form of Web services. It provides three levels of service called Guaranteed, Controlled Load and Best Effort QoS. The behavior of these levels are similar to those defined by the IntServ Architecture ([12]). The G-QoSM framework is composed of - (i) the QoS Grid Server, (ii) an extended version of the Universal Description Discovery and Integration (UDDIe), (iii) a resource reservation manager, (iv) a resource allocation manager and, (v) a policy Grid Service. It uses DSRT ([20]) for reservation of CPU cycles and DiffServ ([9]) for network management. It has facilities for reservation, advanced reservation and admission control. However, the reservation levels are fixed and cannot adjust depending on the load and type of jobs coming to the grid service.

[21] discusses a resource allocation algorithm using reinforcement learning. The algorithm used for reinforcement learning is one step Q Learning. The reward is based on the amount of time used to complete the job by a grid node. [**?**] presents a general methodology for scheduling jobs in soft real-time systems, where the utility of each job decreases as a function of time. It demonstrates a RL based architecture for solving a NP-hard optimal control problem, which is scheduling jobs on multiple CPUs on single or multiple machines.

$$R = sign\{< \rho_i > - \rho_i >\} \tag{1.1}$$

where R is the reward, and $< \rho_i >$ is the average of the response time of jobs completed to date and ρ_i is the response time of the last completed job. This reward structure penalizes a grid node if its response time is larger than the

average response time.

The update equation is

$$Q_{i,t+1} \leftarrow Q_{i,t} + \alpha(R - Q_{i,t}) \tag{1.2}$$

$Q_{i,t}$ is the value of grid node i at time t and α is the learning rate. Therefore grid nodes that take longer time to complete a job have lower value then grid nodes with fast response time. The agent follows a ϵ-greedy strategy, which means that the greedy action will be taken with probability $1-\epsilon$, with the greedy action being taken otherwise. The paper concludes that this Q learning approach does better than an algorithm that simply chooses the least loaded resource. The problem with this simplistic approach is that the arrival of jobs at a resource would be cyclic. A fast grid node would see more jobs coming to it, eventually slowing it down more than the slow resource, which will hardly see any load. As a result its Q value will decrease over time, after which the slow resources will tend to get swamped.

1.8 Aims of this Thesis

This research focuses on the need to provide Quality of Service to grid users in an autonomous manner. Providing QoS in an autonomous manner implies that it should require minimum interference from a system or network administrator in order to meet the QoS requirements laid out in SLAs.

As we stated in Section 1.5, static mechanisms to provide QoS cause resources to be underutilized at a service providers site. In our thesis, we focus on dynamic QoS mechanisms that learn policies to help maximize the utilization of a network or grid service providers resources.

We focus on *Reinforcement Learning* based mechanisms that can observe the load and other conditions at a resource and learn resource allocation policies with-

out the need for external supervision. RL agents reside at each resource node. These agents are responsible for the resource allocation policies. We intend to implement a system where successful completion of jobs within deadlines gives a positive feedback to the agents, and they are penalized if a job fails to finish within deadline. Once the agent learns an optimal policy, they should be able to handle any situation in the network and grid. We aim to design such a system in simulation, as well as have a working implementation on a testbed to assess its viability.

1.9 Organization of this Thesis

This thesis describes how QoS can be achieved using RL based algorithms. The proposal made is evaluated using a grid simulation package known as GridSim, and a testbed running a Globus based grid. Before the actual RL based system is described, we need to briefly introduce these various components. Accordingly, the rest of the thesis is organized as follows - Chapter 2 gives a brief introduction to Reinforcement Learning, the framework which was used for the design of our resource allocation system. It includes descriptions of the Markov Decision Process, states, actions reward policy etc. It also shows how a Cerebellar Model Articulate Controller is used for function approximation. It goes on to describe the RL algorithms used in this thesis, namely Q(λ) and SMART.

Chapter 3 gives a description of the work that was done on GridSim. GridSim did not support any network simulations prior to version 3.1. This chapter details the motivations behind implementing an elementary network stack for GridSim and how the this feature was designed and added to GridSim. We have shown the design for the network elements, and also discussed their implementation details. We have provided class diagrams for the various elements like Packets, Links, Routers etc., that are essential for the network infrastructure in GridSim. We

have also discussed some of the other grid simulation packages, the features they provide, and why they do not fit our needs completely. It is hoped that these will be useful for other researchers wanting to use and extend GridSim.

Chapter 4 discusses our strategies for Reinforcement Learning based resource allocation in detail. Resources can be allocated by provisioning or reservation. The chapter explores resource provisioning on routers and GRs with the use of WFQ and GPS algorithms respectively. It also details how resource reservation can be provided on routers and GRs using rate-jitter schedulers and CPU cycle reservation. We also discuss the design and configuration of the RL system, like the state and action space, reward policies etc.

In Chapter 5, we describe the simulations of the solutions we discussed in Chapter 4. We describe the various simulation experiments we designed and ran on GridSim to benchmark the performance of our solution against currently used static allocation policies. We simulated three distinct scenarios : (i) when only the User broker runs an agent to decide which resource to send the next job to; (ii) when the agents are only running on network routers and grid nodes, adjusting the resource allocation levels according to the policies they learn in real time; (iii) when the agents are enabled on the User brokers, the network routers and grid nodes. We also compare the performance of our algorithm in each case. We continue the chapter with a description of the implementation of our system on a testbed. We detail the design of the agents running on routers and grid resources. These agents also use Reinforcement Learning to determine resource allocation. We show how we used nice levels on Linux to provide CPU provisioning, and a Linux program called `rshaper` to provide bandwidth reservation.

Chapter 2

Reinforcement Learning

2.1 Introduction to Reinforcement Learning

2.1.1 Markov Decision Process

In general reinforcement learning problems can be modelled as Markov Decision Processes (MDP). A MDP consists of

- a set of States $\mathbb{S}$
- a set of Actions $\mathbb{A}$
- a reward function $R : \mathbb{S} \times \mathbb{A} \rightarrow \Re$
- a state transition function $T : \mathbb{S} \times \mathbb{A} \rightarrow \pi(\mathbb{S})$

The state transition function specifies the probability distribution of state transitions, i.e. the probability of moving from state s to state s' given action a, specified as $T(s, a, s')$. The reward function specifies the instantaneous reward when such a transition takes place. A reinforcement learning process that satisfies the Markov property can be modelled as a MDP.

2.1.2 The Markov Property

In a certain process, if the probability distribution of all the future states of the environment only depend on the current state, the process is said to fulfill the *Markovian Property*, and the process itself is known as a *Markovian Process*. This implies that the probability of reaching a state s' from s is always known. Mathematically,

$$P[s_t + 1 = s'|s_t, a_t]$$

is a known value, where $P(s_t, a_t)$ is the probability of event being in state s due to action a at time t.

In reinforcement learning, the agent makes a decision based on a function of the state of the environment. If the given problem is Markovian in nature, then the agent can predict all future states and rewards to be received given an initial state. Thus, Markovian problems provide the best basis for choosing actions. Theorems developed for Markovian problems can be used with non-Markovian processes, but the effectiveness becomes lesser as the problem diverges from the Markovian property.

2.1.3 Reinforcement Learning

Bellman [22] proposed the framework of Dynamic Programming (DP) to solve problems which are amenable to Markovian analysis. Algorithms like value or policy iteration can be used to solve such problems. However, these algorithms require that for every decision made by an agent, the complete state-transition probability is required. In other words, the model of the problem must be known. In problems with large state and action spaces, developing the transition probabilities model can be a hindrance to solving the problem. This is known as the *curse of dimensionality*. To solve such problem, we can use algorithms like the method of temporal differences (TD(λ) [23] or Watkins Q(λ) [24]. These methods

fall in the category of reinforcement learning algorithms.

Reinforcement Learning(RL) (also known as Neuro-dynamic Programming (NDP)) is a popular technique used to learn an action policy i.e. given a certain situation, it is able to decide the ideal action to be taken. An ideal action is one which maximizes a numerical reward signal over a period of time. Agents using the RL algorithm discover by trial and error which *actions* are the most valuable in the *states* that the agents will encounter. For a comprehensive discussion on RL, please refer to [25] and [26].

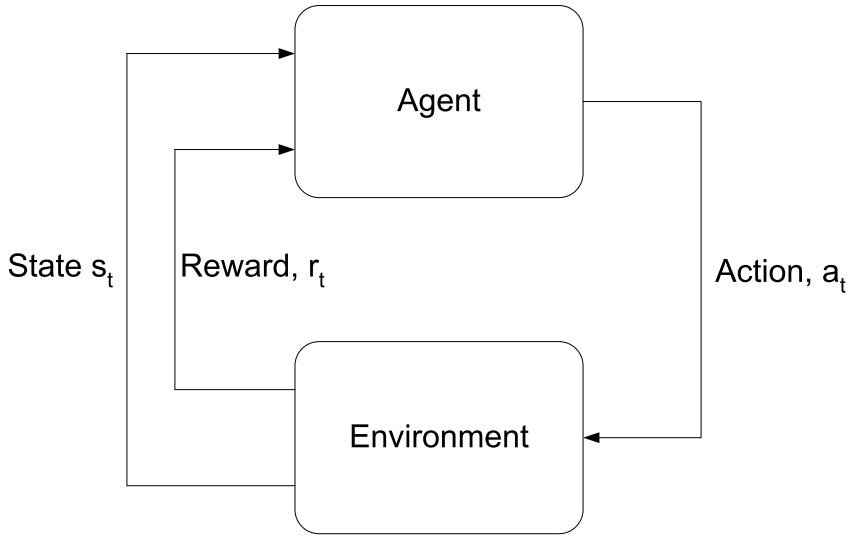

Figure 2.1: Reinforcement Learning Model

In the standard model shown in 2.1, an agent communicates with an environment. At each iteration of the algorithm, the agent gets to know the state s_t from the environment at time t. The agent refers to its current policy (π), and decides an action a_t to be taken. This changes the state of the environment from s_t to s_{t+1}. It also causes a reward to be generated, R_t, which is used as feedback to the agent. The agent learns about the desirability of an action from a particular state in this way. The agent formulates a policy that maximizes the discounted sum of expected future rewards

$$value = E[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + ...] \tag{2.1}$$

2.1.4 State

In every reinforcement learning problem, the environment is always in a certain *state*. A state can consist of any information available to the agent from the environment. For example, if the environment is a chess board, the state of the environment could be the positions of the pieces on the board. States can be directly observed from the environment or constructed using inputs received from the environment. Constructed states can be useful when the agent is interested in the gradient of a feature. In this thesis, the state of the environment at time t is represented as s_t.

2.1.5 Action

At every state, the RL agent has a list of possible actions that it can take from that state. Actions could either be greedy or exploratory. Greedy actions are ones that the agent knows should lead to the future rewards being maximized. Exploratory actions [27] are those that are used to learn more about the environment, in the hope of finding a better policy than the current one. In this thesis, the action taken at time t is represented by a_t.

One way to decide between taking greedy and exploratory actions at each step is known as the ϵ-greedy method. In this method, the greedy action is taken most of the time, but once in a while, an exploratory action is taken with a small probability ϵ.

Another way to explore the state and action space is by using *softmax* action selection. This method is an improvement over ϵ-greedy methods, because the ϵ-greedy selection is as likely to be a bad one as a good one. The softmax algorithm on the other hand chooses actions based on their value estimates. The probability

of an action being chosen is directly proportional to its value estimate, therefore the greedy action has the highest probability of being chosen. A common softmax method uses the Boltzmann distribution.

The probability of choosing action a at time t is

$$p(a) = \frac{e^{Q_t(a)/\theta}}{\sum_{b=1}^{n} e^{Q_t(b)/\theta}} \tag{2.2}$$

θ is a positive parameter called the temperature. The higher the temperature, the more even the probabilities of all actions being chosen. Therefore, an experiment can begin with a high temperature, and the temperature can be lowered as the experiment goes on. This will mean that more learning will take place initially, and as the agent learns more about the optimal policy π^*, less exploratory actions are taken, causing higher returns and system stability.

2.1.6 Rewards

Agents receive reward from the environment for taking actions at each state. The sole purpose of the agent is to maximize the expected value of the rewards in the future, as stated in Equation 2.1. Rewards are generated by the environment, rather than being calculated by the agent itself. By placing the reward assignment outside the agent, it is encouraged to formulate a policy in an environment in which it has imperfect control of the reward outcome.

The rewards generated by environments due to an agent's actions may not be immediate. For example, in a grid environment, an agent could change the reservation levels for certain classes of jobs. This affects the jobs running currently and the jobs arriving in future. Since jobs on grids could be processor and I/O intensive, they could take a long time to complete, and the effect of the agent's action will not be known till some time into the future. Such rewards are known as *Delayed Rewards*, and agents must know how to assign delayed rewards to the

action or series of actions that caused this reward to be generated.

The most important part of assigning reward is to determine which action at which state should the reward be assigned to. This is known as the *temporal credit assignment problem*. If the credit is assigned to an action that did not contribute to the credit being generated, the agent might formulate a sub-optimal policy. A sub-optimal policy is one that does not lead to maximum expected return. One method to overcome this problem is to wait till the end of the experiment, and observe whether the reward is positive or negative, and reward the actions respectively. We will need to iterate this process a few times for proper learning to take place. However, this increases the learning time of the agent, and does not work in problems that do not have a specific ending. There are techniques that adjust the estimated value of a state depending on the current estimated value, immediate reward and the estimated value of the state reached due to the action. These class of techniques are known as *temporal difference methods* ([23]).

2.1.7 Policy

Every agent follows a policy which dictates what actions are to be taken in each state. The policy that the agent follows determines the reward and return that the agent enjoys. By observing the returns from its current policy, the agent tries to continually improve its policy. In our thesis, policy is represented by π, the action to be taken at a state s is represented as $\pi(s)$, and the optimal policy is represented by π^*. The optimal policy is the one that achieves maximum return. Policy can be improved through Policy Iteration or Value Iteration [26].

2.1.8 Function Approximation

The basic method of storing the Q-values of state-action pairs are lookup tables. Lookup tables are simple to implement, but they can only be used when the state and action space is small. However, in problems where the state or action space is

very large or continuous (for example [28]), lookup tables become impractical to implement. Not only do the lookup tables require large amounts of memory, they are also unable to generalize the learnt values. *Generalization* refers to the generation of similar outputs for similar input. Generalization also allows agents to learn faster since they do not need to explore the entire state and action space before they can produce a meaningful action for a given state-action pair. Structures that provide such generalization are known as *Function Approximators*.

There are several techniques which can provide function approximation. We can use neural network methods like Multi-Layer Perceptrons (MLP) or Radial Basis Functions (RBF). The method we have chosen for function approximation in our experiments is a coarse coding technique known *Cerebellar Model Articulation Controller* (CMAC). The main advantage of CMACs over RBFs and MLPs is that they have lower computation requirements ([29]).

CMACs work by using quantizing functions and resolution elements. Each dimension of the Q-value is mapped to K different quantizers, each of which has N resolution elements. Each quantizing function has a value associated with it, and each resolution represents a fraction of the value of the quantizer. When two inputs to the CMAC are close, they map to quantizers and resolution elements that are close, and therefore have values that are similar. A detailed description of CMACs can be found in Chapter 3 of[29].

For a CMAC with K quantizing functions, the number of storage elements required is

$$S = K \ \Pi_{i=1}^{j} \ N_i \tag{2.3}$$

where N_i is the number of resolution elements in quantizer i. Choosing a larger number of quantizers and resolution elements increases the accuracy of the CMAC, but also increases its memory requirements.

The major advantage of using CMACs is that looking up the value of a single

element only requires K lookups, where K is the number of quantizers. This is significantly lesser than the case of MLPs where all weights have to be computed to lookup a single value.

2.2 Solutions to Reinforcement Learning

In our problem of providing Quality of Service in Grid Environment, we experimented with two different methods that can be used to solve reinforcement learning problems. The first of these is called Watkins Q-Learning algorithm [30] , and the second one is known as Semi-Markovian Average Reward Technique (SMART) [31].

2.2.1 Watkins' Q-Lambda

Watkins' $Q(\lambda)$ is an off policy Temporal Difference (TD) learning method [24]. It combines one-step Q-Learning with eligibility traces. Both these terms are explained below.

One-step Q-Learning

In one step Q-Learning, the agent uses the following update rule

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[R_{t+1} + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{2.4}$$

The $Q^\pi(s, a)$ function is the perceived value of taking action a at state s under policy π. The higher the value of $Q^\pi(s, a)$, the more desirable that action in that particular state. $Q^\pi(s, a)$ is basically the expected return when starting from state s, taking action a and following policy π thereafter. The Q-value of a state-action pair is learnt from experience. If the agent follows policy π and maintains an separate average for actions from each state, the average will converge to the Q-value for that state. In a reinforcement learning problem, the objective of the

actions should be to maximize the return. Achieving maximum returns indicates that the agent is following the optimal policy π^*.

Equation 2.4 above is known as one-step Q learning because the Q-value of a state-action pair is only updated with the Q-value of the next state-action pair visited. Steps taken by the agent do not affect state-action pairs visited more than one epoch ago. It is also an offline method because the next step Q-value being used to update $Q(s_t, a_t)$ is not $Q(s_{t+1}, a_{t+1})$. Rather it's $max_a Q(s_{t+1}, a)$. This means is that the to update the Q-value of a state-action pair reached at time t we do not use the next state-action pair reached, but we use the maximum of the possible Q values of the state-action pairs that could be reached from (s, a). This method has been shown to achieve faster convergence than using the Q value of the next state-action pair (s_{t+1}, a_{t+1}). The policy decides the next action to be taken, but the updates to the Q-values are independent of policy. In effect, the learning is greedy, while the policy is exploratory.

The term $[R_{t+1} + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$ in equation 2.4 is known as the *Temporal Difference* (TD) error. The TD error is an indicator of whether the action that was taken leads to better Q values for a state-action, or worse. A positive TD error indicates that the tendency to select action a_t at state s_t should be increased, a negative TD error indicates the opposite.

Eligibility Traces

In most real-life reinforcement learning problems, there is a delay between taking an action and the generation of the reward or penalties. Eligibility traces [32] are a way to overcome the problem of which state-action pair the reward should be assigned to. An example of the use of eligibility traces is Tesauro's backgammon playing reinforcement learning agent [28].

Eligibility traces are akin to maintaining a trace of all the events that have happened that lead to up to the visit of a state or action. By keeping this trace,

all the state-action pairs that led to this TD error can be rewarded or penalized. This is desirable because the generated TD error is not just due to the immediate action that caused the error; the path of state-action pairs traversed is responsible for the generation of the current TD error. Thus the eligibility trace helps to assign rewards or penalties to all the state-action pairs responsible for this TD error.

Two kinds of eligibility traces are generally used, *accumulating traces* and *replacing traces*. In accumulating traces, the weight assigned to the state-action pair is increased each time that state-action pair is visited. In replacing traces, the weight of the trace for a state-action pair is set to 1 if a state- action pair is visited, and remains at one even if it is visited multiple times before a TD error is generated. Replacing traces are more efficient than accumulating traces because they only need to maintain a binary map for all the stat-action pair. Accumulating traces require an array of integer or fractional values, which requires more memory. In problems with very large state and action spaces, the memory requirements of accumulating traces might be prohibitive. [32] showed that replacing traces improve convergence in a "Mountain-Car" experiment, a problem in which the state and action space is continuous. Replacing traces were used in this study due their efficiency and better performance.

Replacing traces are defined in the following way -

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & s \neq s_t \\ 1 & s = s_t \end{cases} \tag{2.5}$$

In equation 2.5, the eligibility trace for all states are decayed by a factor $\gamma\lambda$. The state being visited has its eligibility trace set to 1. γ is the discount rate and λ is called the *trace-decay parameter*. The use of trace-decay parameter ensures that the states that were visited earlier than when the TD error was generated, receive a lesser share of the credit or blame. This is sensible because states that

are closer predecessors in time are more responsible for the agent reaching a state where the current TD error was generated.

$\mathbf{Q}(\lambda)$

Merging the concepts of one step Q-learning and eligibility traces gives us Watkins Q(λ).

Watkins Q(λ) uses equations 2.6 and 2.7 to update Q values for state-action pairs. For recording the eligibility trace, when a state is visited, its eligibility trace is set to 1, and all others are decayed by $\gamma\lambda$. Whenever an exploratory action is taken instead of a greedy one, all the trace values are set to 0.

$$\delta \leftarrow R + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t) \tag{2.6}$$

For all s,a:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\delta e(s_t, a) \tag{2.7}$$

However, in our simulations and implementation, the value functions are calculated by a CMAC. Therefore, the RL agent only calculated the TD errors according to Equation 2.6, and then uses that value to update the features of the CMAC.

2.2.2 SMART

SMART (**S**emi **M**arkovian **A**verage **R**eward **T**echnique) ([31]) is a RL technique to solve Semi-Markov Decision Problems (SMDP).

Semi Markov Decision Problems

Markov Decision Problems require that a decision is made every time step. When the sojourn times between actions are drawn from general probability distributions, the problems can be modeled as Semi- Markov Decision Processes.

SMART

The Bellman optimality equation for SMDPs can be stated as follows ([33]) : There exists a scalar χ^* and a value function V^* satisfying the system of equations

$$V^*(s) = R(s,a) - \chi^*\tau(s,a) + \sum_{s' \in \mathbb{S}} P_{ss'}(a) \max_{a' \in \mathbb{A}_{s'}} V^*(s',a), \forall s \in \mathbb{S} \qquad (2.8)$$

where $\tau(s,a)$ is the average sojourn time of the SMDP in state s under action a, $R(s,a)$ is the immediate reward obtained by the action, $P_{ss'}(a)$ is the transition probability of going from state s to state s' given action a, and the greedy policy π^* formed by selecting actions that maximize the right hand side of the above equation is optimal.

Jobs in grids can take indefinite amount of time. The time not only depends on the number of grid jobs, it will also change according to the local load at the grid node. The problem we are trying to solve is non-deterministic. Thus constructing a model for transition probabilities will be an extremely hard task. Consequently deriving a value iteration algorithm will also be difficult. Since the amount of time taken by jobs is indefinite, the sojourn times are also not fixed. Therefore this problem can be modelled as a SMDP.

$V(s,a)$ in Equation 2.8 can be calculated by SMART online using equation 2.9 shown below.

$$\begin{aligned} V_{new}(s,a) &= (1-\alpha_m)V_{old}(s,a) \\ &+\alpha_m \left(R(s,s',a) - \mathcal{AR}_m\tau(s,s',a) + \max_{a'} V_{old}(s',a') \right) \end{aligned} \qquad (2.9)$$

where $V_{new}(s,a)$ is the new calculated value of action a in state s at the m^{th} decision epoch, α is the learning rate at that epoch for updating the value of the state-action pair, s' is the new state that the system moves into due to action a, and a' is the action taken from state s'. $R(s,s',a)$ is net reward earned between

two successive decision epochs, starting in state s, ending in state s' due to action a. $\mathcal{AR}_m$ is the average reward rate, and it is updated as shown in equation 2.10.

$$\mathcal{AR}_m = (1 - \beta_{m-1})\mathcal{AR}_{m-1} + \beta_{m-1}\frac{T(m-1)\mathcal{AR}_{m-1} + R(s', s, a)}{T(m)} \tag{2.10}$$

T(m) is the total time spent in all visited states until the m^{th} decision period and β_m is a learning rate parameter.

The learning rates α_m and β_m, and the exploration rate ϵ_m are decayed slowly to 0 using the Darken-Chang-Moody search-and-converge algorithm [34].

2.3 Advantages of Reinforcement Learning

A reinforcement learning based solution allows us to combine various states, actions and rewards into a policy. This is advantageous in problems where the action may cause an indirect change in state, and therefore the action has no direct control over the returns from the environment either. Often problems are non-deterministic i.e. the probability of reaching a specific state due to an action taken in another state need not be 0. In fact, in real world situations, non-deterministic problems are more common than deterministic ones. Reinforcement Learning also frees us from having to study the state transitions in detail in order to build a model of the problem.

Chapter 3

Design of Network Elements in GridSim

In this chapter, we detail the work we did in designing network extensions for GridSim. For our simulation purposes, we required a package that provides support for grid modelling, network support and flexible and easy to modify for our purposes. We were not able to find any simulation package with all these requirement, but found GridSim to be easily modifiable to meet our needs. Since the only thing missing from GridSim was a network stack, we wrote a network layer for GridSim. This chapter is mostly derived from [35]. It has been reproduced here since the design and implementation of GridSim is essential to understanding the simulations we conducted in Chapter 5.

3.1 Introduction to GridSim

Grid computing has emerged as the next-generation parallel and distributed computing methodology that aggregates dispersed heterogeneous resources for solving various kinds of large-scale parallel applications in science, engineering and commerce [36]. In order to evaluate the performance of a grid environment, experiments need to be *repeatable* and *controlled* [37] , which are difficult due to grid's

inherent heterogeneity and its dynamic nature. Additionally, grid testbeds are limited and creating an adequately-sized testbed is expensive and time consuming. Grids also need to handle different administration policies at each resource. Due to these reasons, it is easier to use simulation as a means of studying complex scenarios.

GridSim ([38]) is a Java based discrete-event Grid simulation toolkit. It supports the modelling and simulation of heterogeneous resources, users and application models. The GridSim toolkit has been applied successfully to simulate a Nimrod-G [39] like grid resource broker and to evaluate the performance of deadline and budget constrained cost and time optimization scheduling algorithms.

3.2 The Need for Network Simulation in GridSim

Versions of GridSim 3.0 and earlier did not support designing a network architecture to connect the various users, resources and other entities that may be required (Grid Information Service for e.g.). In the real world though, Grid computing technologies are increasingly being used to aggregate computing resources that are geographically distributed across different locations. Commercial networks are used to connect these resources, hence a simulation that does not take into account the effect of network links and routers will be incomplete.

Support was added to GridSim to simulate various network elements like links and routers. GridSim now has the ability to

1. allowing users to create a network topology,
2. packetizing a data into smaller chunks for sending it over a network,
3. generating background traffic, and
4. incorporating different level of services for sending packets.

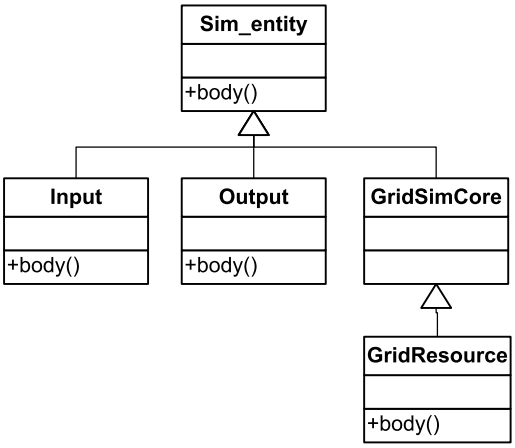

Figure 3.1: A Class Diagram Showing the Relationship between GridSim and SimJava entities

The flow of information among GridSim entities happens via their *Input* and *Output* (I/O) entities. Upon creating a GridSim entity with a specified bandwidth, it automatically creates both instances of class `Input` and `Output`, and links them to this entity. Hence, sending a data must go through to a sender's *Output* entity before going into a recipient's *Input* entity for collection.

3.3 Design and Implementation of Network in GridSim

GridSim is based on SimJava2 ([40],[41]), a general purpose discrete-event simulation package implemented in Java. In SimJava, each simulated system (e.g. resource and user), that interacts with others, is referred to as an entity. An entity runs in parallel in its own thread by inheriting from the class `Sim_entity`, while its desired behavior must be implemented by overriding a `body()` method.

SimJava requires each entity to have two ports for communication: one for sending events to other entities, whereas the other port is used for receiving incoming events. In GridSim, this is done via class `Input` and `Output`.Both classes have their own `body()` method to handle incoming and outgoing events respectively.

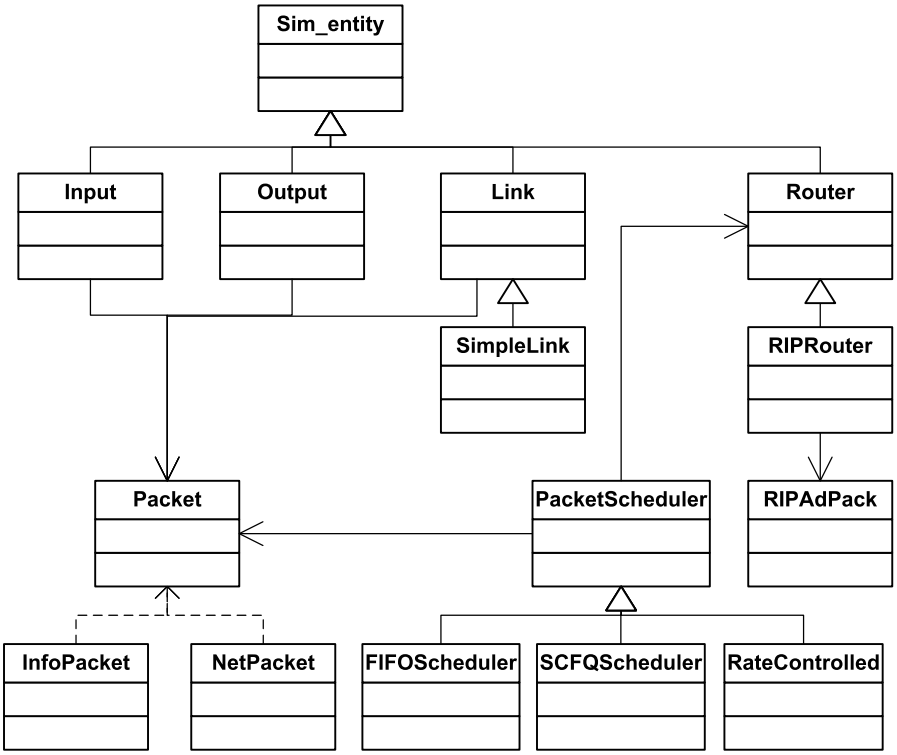

Figure 3.2: A Class Diagram Showing the Relationship between GridSim and SimJava Entities

Similar to SimJava, GridSim entities must inherit from the class `GridSimCore` and override a `body()` method. The relationship between `Sim_entity` and GridSim classes is shown in Figure 3.1. In a class diagram, attributes and methods are prefixed with characters + indicating access modifiers public. The class `GridSimCore` does not have the `body()` method because it is not necessary since its subclasses will override the method.

The flow of information among GridSim entities happens via their *Input* and *Output* (I/O) entities. Upon creating a GridSim entity with a specified bandwidth, it automatically creates both instances of class `Input` and `Output`, and links them to this entity. Hence, sending a data must go through to a sender's *Output* entity before going into a recipient's *Input* entity for collection.

The use of separate entities for I/O provides a simple mechanism for a GridSim entity to communicate with each other, and allows modeling of a communications

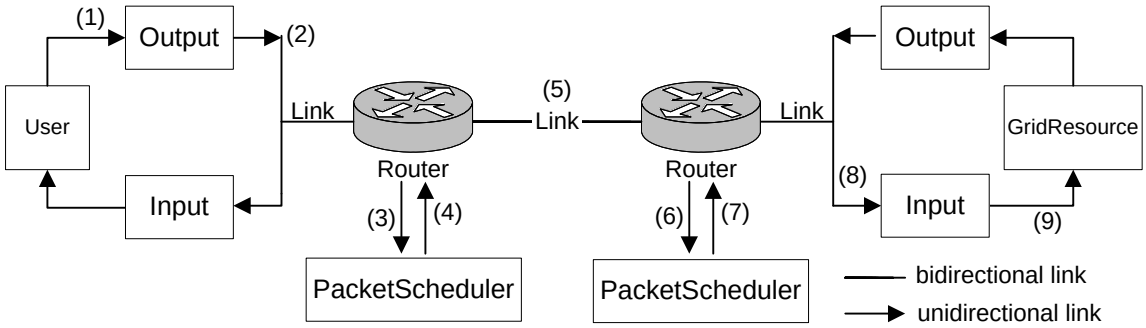

Figure 3.3: Interaction among GridSim Network Components

delay ([37]). In addition, this existing design provides a clean interface between the network entities and others. Therefore, most of the changes were incorporated into class `Input` and `Output` for transparent and minimal modification to the existing code.

The new addition to the existing network architecture allows GridSim entities to be connected using links and routers, with different packet scheduling policies for realistic experiments as shown in Figure 3.3. Detailed explanation of this figure will be explained later in Section 3.3.3. The network architecture has also been designed to be extensible and backwards compatible with existing codes written on older GridSim releases.

3.3.1 Network Components

Important addition to the existing GridSim network architecture are link, router, packet, packet scheduler and background traffic generator components. The relationships among these network components in Unified Modeling Language (UML) notations [42] are depicted in Figure 3.4 and 3.5.

Link

A link in GridSim is represented as an abstract class `Link` for extensibility. `SimpleLink`, a subclass of `Link` as shown in Figure 3.4 (a), requires information like the propagation delay, bandwidth and Maximum Transmission Unit (MTU) for packet

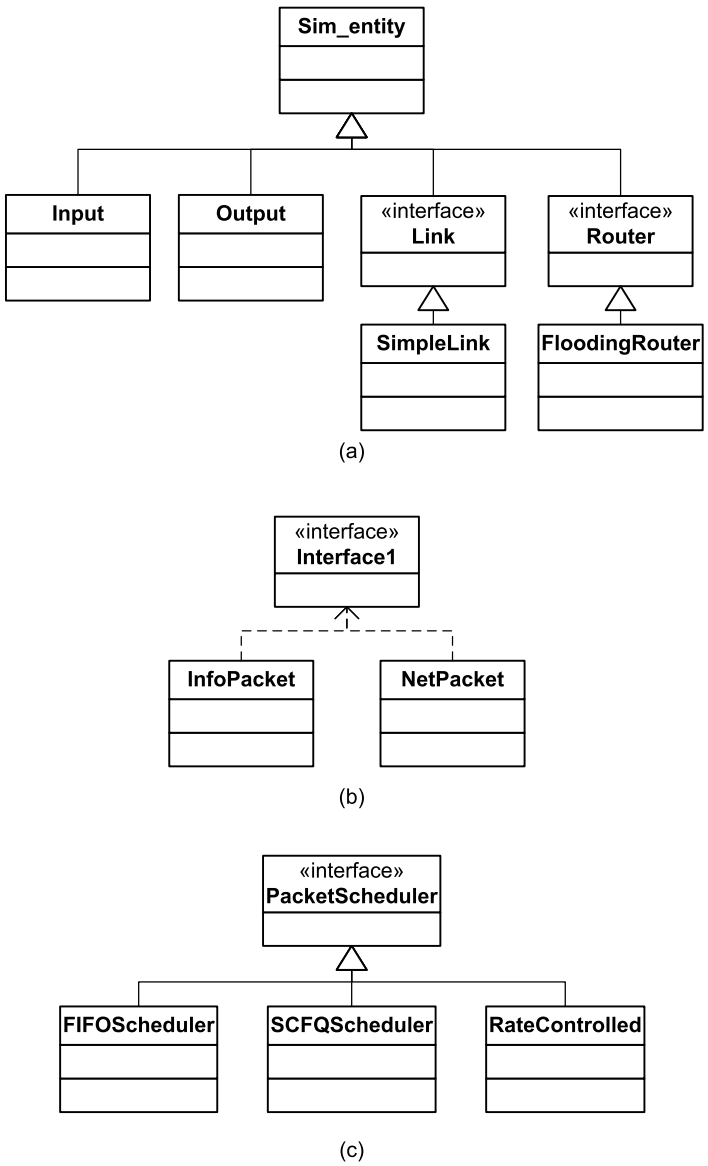

Figure 3.4: Generalization and Realization Relationship in UML for GridSim Network Classes

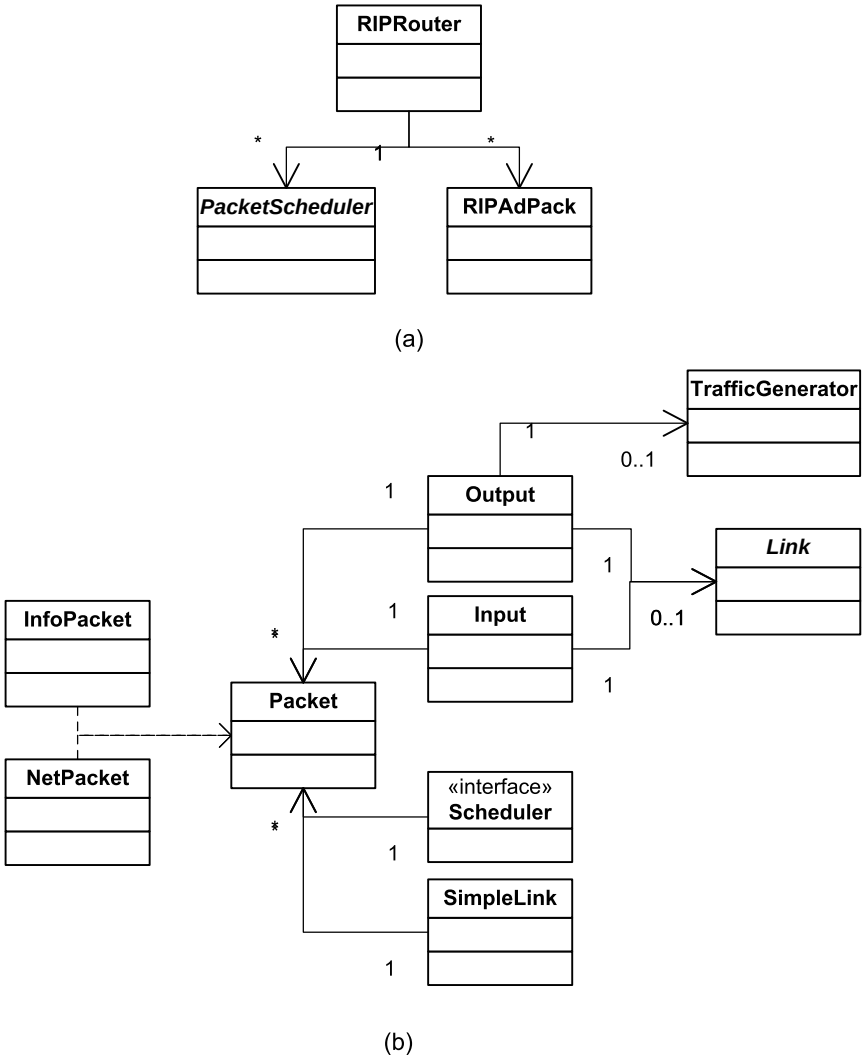

Figure 3.5: Association Relationship in UML for GridSim Network Classes

delivery.

Input and Output

When GridSim entities want to send a data over the network, each of them has *Input* and *Output* (I/O) entities attached to it, as previously mentioned. The *Output* entity is responsible for splitting the data into MTU sized packets, whereas the *Input* entity is accountable to collate the different packets in a stream altogether, and send them as one piece of data to the GridSim entity. In addition, these I/O entities act as a buffer to hold the packets until a link is free.

Router

A router in GridSim is represented as an abstract class `Router` for flexibility as shown in Figure 3.4 (a). Therefore, this design allows a subclass of `Router` in determining the forwarding table at the start of the simulation, and implementing any routing algorithms.

Routing can be done using static tables or dynamic methods, such as Routing Information Protocol (RIP) [43] and Open Shortest Path First (OSPF) [44]. An implementation of a router in class `FloodingRouter` uses a flooding algorithm to setup its forwarding tables automatically. Since routers and other GridSim entities can not be created and added after the simulation has started, the flooding algorithm is a sufficient method to setup a router's forwarding tables.

Packet

A network packet in GridSim is represented as an interface class `Packet` as shown in Figure 3.4 (b). Currently, there are two classes that belongs to this category, i.e. `NetPacket` and `InfoPacket`. A `NetPacket` class is used to encapsulate data passing through the network, whereas class `InfoPacket` is devoted to gather network information during runtime which is equivalent to Internet Control Message

Protocol (ICMP) [45] in physical networks.

Packet Scheduler

A packet scheduler is responsible for deciding the order in which one or more packets will be sent downlink. Implementing a packet scheduler requires extending from class `PacketScheduler` as depicted in Figure 3.4 (c).

Two implementations of a packet scheduler are provided in GridSim, i.e. class `FIFOScheduler` and `SCFQScheduler`. The class `FIFOScheduler` uses a simple First In First Out (FIFO) policy, whereas the class `SCFQScheduler` adopts a variation of Weighted Fair Queuing (WFQ) [46], called Self Clocked Fair Queuing (SCFQ) [47] policy, which will be discussed next.

3.3.2 Support for Network Quality of Service & Runtime Information

Jobs on grids may have different requirements with respect to bandwidth and latency. Systems like fire or earthquake detection require low latency and reliable delivery. Other jobs like protein folding experiments require high processing power, and may tolerate some network errors. Also, in some cases, grid resource providers may wish to charge for priority access to their resources. Thus grid resource providers need mechanisms to provide users with different Quality of Service (QoS) for using their networks [9]. In order to support this functionality, every packet in GridSim contains a Type of Service (ToS) attribute with a default value of zero weight. This attribute will be used by routers or packet schedulers to provide a differentiated service to heterogeneous links or connections for incoming packets. In GridSim, class `SCFQScheduler` can be configured with different weights. Packets belonging to a class with higher weight receive higher priority according to the SCFQ algorithm.

GridSim also supports requesting network status during runtime, such as num-

ber of hops to destination, round trip time (RTT), bottleneck bandwidth and all bandwidths that a packet has traversed for current or future simulation time. This feature is similar to an ICMP ping message. The result is captured inside class `InfoPacket`.

To enable this functionality, a GridSim entity must use either blocking or non-blocking method calls from class `GridSimCore`. A blocking call requires to use only a `pingBlockingCall()` method, where it waits for a result to come back while preventing other entity's activities. In contrast, a non-blocking call needs to use a combination of `ping()` and `getPingResult()` methods while doing something else in between. Both `pingBlockingCall()` and `getPingResult()` method return an object of class `InfoPacket`.

3.3.3 Interaction among GridSim Network Components

When a simulation starts, routers send out advertisement packets to all neighboring router, advertising any other GridSim entities they are connected to. Later on, the neighboring routers adjust their forwarding tables upon receiving these packets. Then, they forward the packets to all neighboring routers except the source. Depending on the complexity of a network topology and number of GridSim entities created, this process might take a while.

Once the forwarding tables have been completed, a GridSim entity, named *User* from following an example shown in Figure 3.3, can start sending jobs to a *GridResource* entity. Each GridSim entity has I/O entities attached to it that act as a buffer. Therefore, when a job is to be sent out by a *User* entity, it is first buffered at the *Output* entity (step 1). Here, the job is split into multiple packets if it is larger than the MTU of a link connected to the *Output* entity. The packets are then given sequence numbers, en-queuing in a buffer, and sent to the router down the link one by one. The link takes the packet, delays it by the propagation delay specified, and dequeues it at the other end (step 2).

Routers receive the packet from the link, and decide the packet scheduler that the packet should be sent to (step 3). If the outgoing interface has a MTU less than the packet size, it splits the packet into smaller ones, similar to what *Output* entity does. Next, these packets are enqueued at the packet scheduler. The packet scheduler uses its own algorithm, such as FIFO or WFQ to decide the order in which the packets should be dequeued (step 4). When a link attached to the packet scheduler is free, the router dequeues one packet from the packet scheduler, and sends it down the link (step 5). Similar approach is required if the other end of the link is another router entity (step 6–8).

When the final link is traversed and the packet reaches the *GridResource* entity, all packets in a sequence are collated back together into the job (step 9). This is done by the *Input* entity. The job is then passed to the *GridResource* entity for processing. Once processing is complete, the *GridResource* entity passes the completed job to its *Output* entity, which follows a similar path until it reaches the *Input* entity that created this job.

The current protocol used for sending packets is a datagram oriented protocol, which is similar to User Datagram Protocol (UDP). There is no support for acknowledging each packets and packet reordering. Since there is no support for recovering lost packets, I/O buffers are considered to be unlimited in order to ensure no packets are lost.

3.4 Related Work

Simulation is very much used in the networking research area. Examples of such simulators include NS–2 [48], DaSSF [49], OMNET++ [50] and J–Sim [51]. Though their support for network protocols is extensive, they are not targeted at studying grid computing. This is because simulating grids requires modeling the effects of scheduling algorithms on grid resources and investigating user's QoS

Table 3.1: Listing of Network Functionalities for Each Grid Simulator

Simulation Tool	Routing Table Entry	Type of Transport Protocol	Data Packetization	Runtime Network Status	Network QoS
GridSim	Automatic	a datagram oriented protocol similar to UDP	Supported	Supported	Supported
MicroGrid	Automatic	TCP and UDP	Supported	Supported	Not supported
SimGrid	Manual	TCP	Not supported	Supported	Not supported
OptorSim	Manual	Not supported	Not supported	Not supported	Not supported

requirements for application processes. In addition, we believe simulating TCP and UDP connections are sufficient to model a real world behavior, because grid users are mostly interested in finding out RTT and available bandwidth of a host. Therefore, these network simulators perform other complex functionalities which are not needed in simulating a grid computing environment.

There are some tools available, apart from GridSim, for application scheduling simulation in Grid computing environments, such as Bricks [52], MicroGrid [53] [54], SimGrid [55] [56], and OptorSim [57]. All of these simulators also have an underlying network infrastructure, with the ability to simulate realistic experiments by using background traffic. Differences among the grid simulators, except for Bricks, in terms of network functionalities and features are highlighted in Table 3.1. Note that for Routing Table Entry column, an automatic entry means filling in a router's forwarding table automatically during runtime. In contrast, a manual entry means filling in the forwarding table by reading from an external file that defines a router's connection with others, or by manually entering the information into the table.

Bricks [52] is able to specify a network topology, bandwidth, throughput and

variance of the throughput over time. The background traffic functionality is modeled by using a probabilistic distribution, which is similar to GridSim. However, at the time this article is being written, this package is not available to download from its website [58]. As a result, we are not able to compare it with our work in more details. Therefore, it is not included in Table 3.1.

MicroGrid [53] [54] allows complex network modeling, such as transport and routing protocols, and large-scale experiments since it is based on DaSSF [49]. Hence, in terms of network capabilities, MicroGrid is the most complete of all grid simulators. However, it is actually an emulator, meaning that actual application code is executed on the virtual grid modeled after Globus [59].

SimGrid [55] [56] has a good network infrastructure that supports Transmission Control Protocol (TCP) transport protocol for a reliable service. It also models background traffic by reading from a trace file generated by Network Weather Service (NWS) [60]. NWS is used to monitor current available bandwidth between two machines over the network. However, SimGrid does not make any distinction between a job computation and a data transfer, since they are modeled as a resource performing a specific task. Therefore, it does not support data packetization. In addition, requesting network status functionalities during runtime in SimGrid are limited to latency and bandwidth of a link. In contrast, GridSim reports more network information than SimGrid, such as number of hops to a destination and RTT as mentioned in Section 3.3.2.

OptorSim [57] has a very simple network infrastructure compared to other simulation tools, since it does not support routing and transport protocol nor data packetization. The background traffic functionality is modeled by using a Landau distribution only. In addition, simulating with background traffic requires a configuration file that describes a network topology in a matrix format.

From the above discussion and Table 3.1, GridSim incorporated QoS into a network for scheduling packets, which are not supported by other grid simulators.

In addition, GridSim provides a good set of network functionalities and features, which some of them are not supported in the other grid simulators.

3.5 Conclusion to GridSim

Network serves as a fundamental component in grid computing since resources and users are connected over a network topology with shared bandwidth. Previously, GridSim does not have the ability to specify a network topology nor the functionality to connect resources through network links in the experiment. In this work, modifications into an existing network architecture have been incorporated into `GridSim ver3.1` to address the above problems.

With the addition of this network functionality, users can study the effects that both the network topology and grid resources can have on their jobs. This paper explores the various types of network elements in GridSim like routers, links, packet schedulers; and how they can be extended to add more functionalities. Moreover, GridSim has new exciting features such as generating background traffic during an experiment, requesting network information during runtime and providing differentiated service for packets based on users' Quality of Service (QoS) requirements. These features help make GridSim a comprehensive package to simulate a realistic grid environment.

Software Availability

The latest GridSim toolkit with source code and examples can be downloaded from the following website:

`http://www.gridbus.org/gridsim/`

Chapter 4

Reinforcement Learning based Resource Allocation

In this chapter, we explore the life cycle of a grid job in detail. We discuss the mechanisms to provide QoS support for provisioning and reservation at the network and processing levels. We also discuss the role of Reinforcement Learning (RL) agents at the resources, how they are configured and their effects on the performance of jobs.

4.1 Life-cycle of a Grid Job

In a typical grid usage scenario in our thesis, a user sends his job to a local broker. The User Broker (UB), decides where to submit the job depending on the

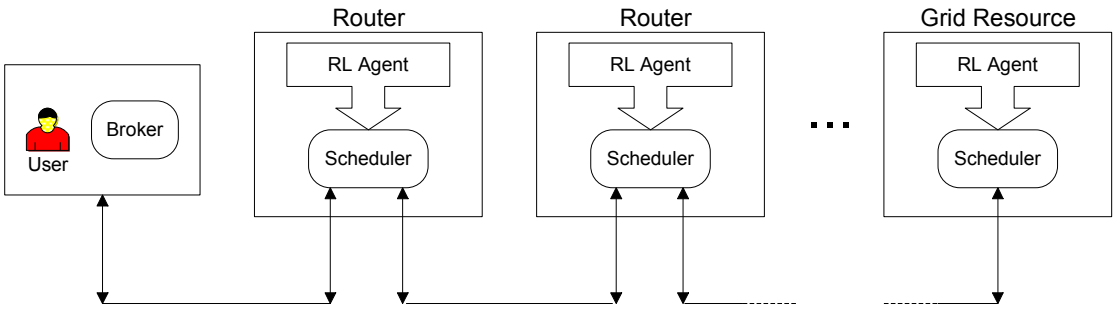

Figure 4.1: Flow of a Grid Job

requirements of the job, availability and load conditions on grid nodes, budget and time constraints etc. The UB also runs an agent within itself, which can decide which Grid Resource (GR) should process the job. The UB can also negotiate or decide which class the job belongs to. Once the grid node is chosen, the UB connects to the GR and sends the job to it. The data passes through a few routers in one or more ADs before reaching the GR. In our framework, show in Figure 4.1, each router and the GR have an agent residing in them.

At each router, the data packets are received at the incoming interface, and put into a queue. The order of dequeuing is decided by a scheduler residing at the router. In our experiments, the scheduling algorithm was one among FIFO, WFQ and a Rate-Jitter scheduler. WFQ and Rate-Jitter schedulers provide QoS support, by providing differential service depending on the class of a job. The agent running at each router can adjust the service levels provided to each class of jobs.

When the job reaches a GR, it is enqueued or processed along with other jobs that might be running on it. A scheduler running at the GR decides the order of processing jobs, as well as the kind of service it receives. In the case of a FIFO scheduler, one job is dequeued from the head of the queue, and it receives exclusive service from the GR. In a parallel scheduler with only Best Effort (BE) service, jobs are never enqueued. All the incoming jobs are run at the same time, with no differentiation in service. Other scheduling schemes may support providing different service levels depending on the class of the job. Each GR also runs an agent, which can adjust the service levels if the scheduling policy supports providing QoS.

Once the job is complete at the GR, it is sent back to the UB. It is the function of the UB to verify whether the job was received back within deadline. As discussed in Section 1.2, the deadline of a job is defined by its class. If the job is finished within deadline, the UB generates a positive reward, and notifies the first router

of the reward along with information about which GR processed the job. If the processed job is only returned after the deadline, a negative reward is generated, and the same action is taken. The UB also updates its value functions with the generated reward. This affects its decisions about choosing GRs for future jobs of the user.

The router that receives the reward and GR information from the User Broker, updates its own agent with it. The agent adjusts the service level parameters of the scheduler at the router. This adjustment depends on the size of the reward received and the class of the job. It also sends this information to the next router in the path. In this manner, each router which processed data packets for that job receives the reward or penalty signal, until the information reaches the GR that processed the job. In order for this to work, all data packets and the reward information for a single job must pass through the same path.

When the GR receives the reward information, it updates its agent with the information. Similar to the agents running on routers, the agents adjusts the service level parameters of the scheduler running on the GR, depending on the size of the reward and the class of the job. This is how a job circulates through a grid system, and the agents learn and adjust QoS parameter to maximize the reward they receive. The net accumulated reward could have monetary offsets, in order to encourage administrator to run agents on their resources to maxima's their profit.

4.2 Network QoS

Service differentiation on the network can be done through bandwidth provisioning or reservation. Service differentiation allows certain classes of traffic to have better service through higher priority access to network resources than others. By queuing all the packets belonging to the same class at the same queue, we ensure

that all flows of the same class receive similar treatment by the scheduler.

In our experiments, we have implemented WFQ to provide provisioning and Rate-Jitter scheduling to provide reservation facilities on the routers.

4.2.1 Bandwidth Provisioning via Weighted Fair Queuing

In this thesis, provisioning is done using WFQ ([61]). WFQ is a fair and efficient way of provisioning bandwidth. The total bandwidth available is distributed among all flows according to the weight assigned to each flow. The higher the weight of a stream, the higher its proportion of bandwidth.

WFQ is a variation of the Generalized Processor Sharing (GPS) algorithm ([62]). A GPS server is a work conserving scheme which operates at a fixed rate μ. If the weights assigned to each flow i is ϕ_i, then a server is said to be GPS if equation 4.1 is satisfied.

$$\frac{TS_i(\tau, t)}{TS_j(\tau, t)} \geqslant \frac{\phi_i}{\phi_j} \tag{4.1}$$

where $TS_i(r, t)$ is the amount of traffic served for flow i in the interval $[\tau, t]$. Both flows, i and j must have backlogged data in the interval $[\tau, t]$. A flow i is said to be backlogged at a server at time t, if there is some data from i queued at the server.

GPS is unimplementable because it assumes that data is perfectly divisible. However, in computer networks, data is always sent in quantized packets. WFQ is a scheme which works with packets of data and approximates the behavior of GPS.

WFQ can provide some statistical guarantees to flows. For a backlogged stream, WFQ provisions bandwidth according to equation (4.2). In equation (4.2) μ_i is the amount of bandwidth provisioned to flow i, ϕ_i is the weight assigned to flow i and n is the total number of flows. μ is the bandwidth of the link.

$$\mu_i = \frac{\phi_i}{\sum_{j=1}^{n} \phi_j} \mu \tag{4.2}$$

The Parekh-Gallagher theorem [62] can provide us a bound on the maximum end-to-end delay. Consider a flow i regulated by a leaky bucket regulator, so that the number of bits sent in time $[t1, t2]$ is bound by $\mu_i(t2 - t1) + \sigma_i$, where μ_i is the minimum bandwidth allocated to the flow among all schedulers it flows through. If this flow passes through K schedulers, and the kth scheduler has a rate $\mu(k)$ then

$$D_i = \frac{\sigma_i}{\mu_i} + \sum_{k=1}^{K-1} \frac{\Gamma}{\mu} + \sum_{k=1}^{K} \frac{\Gamma}{\mu(k)} \tag{4.3}$$

where D_i is the delay bound of flow i and Γ is the size of the largest packet in the network.

Thus, we can see that WFQ can provide some guarantees on bandwidth and delay of a flow. However, the problem with WFQ is that in order to decrease the delay bound of a flow, its bandwidth allocation needs to be increased. Thus a stream which requires low delay needs to be allocated a large amount of bandwidth, even if it does not require the large bandwidth. The delay requirements and bandwidth allocations are coupled in WFQ.

WFQ is a complex algorithm that requires computing a finish time for each packet. The finish time of a packet in a WFQ scheduler is shown in Equation 4.4, where F_i^p is the finish time of packet p of flow i, ϕ_i is the weight attached to flow i and $v(a_i^p)$ is the virtual clock time for that flow. Therefore WFQ requires maintaining a virtual clock for each flow. Also, the virtual time for a flow only increases if there are backlogged packets for that queue. It needs to be paused and remembered when the there are no packets for that flow type. Therefore, WFQ requires the scheduler to maintain a separate virtual clock for each connection .

$$F_i^p = \frac{L_i^p}{\phi_i} + max[F_k^{p-1}, v(a_i^p)] \tag{4.4}$$

Self-Clocked Fair Queuing (SCFQ) ([47]) is a simple to implement approximation of WFQ. In SCFQ, the virtual time increases during busy periods for each flow. When there are no backlogged packets for a flow, its virtual clock is reset to 0. Therefore, the virtual clock does not need to be maintained for each connection. In our experiments, we have used SCFQ instead of WFQ.

The agents running at the routers are allowed to modify the weights that are assigned to each class of traffic. In this manner, they are able to modify the service parameters that a class of jobs experience. Assigning high weights to a class leads to shorter queuing times, and also higher bandwidth utilization.

4.2.2 Bandwidth Reservation via Rate-Jitter Scheduling

We used a rate-jitter regulator ([63], [64]) to provide means to reserve bandwidth at the routers. A rate-jitter regulator is a non-work conserving scheduler . A non-work conserving scheduler is one that may be idle (i.e. not serve packets) even when the downstream link is idle, and packets are queued at it. This helps to constrict the rate of flows, and results in downstream flows being more predictable and less bursty.

In a rate-jitter regulator, the eligibility time of a packet is defined with reference to the eligibility time of the previous packet. If the maximum rate for flow i is μ_i, then the eligibility time of a packet is defined as

$$\tau_{in} = max(\tau_{i(n-1)}, \Lambda_n) + \frac{\Gamma_n}{\mu_i} \tag{4.5}$$

τ_{in} is the eligibility time of packet n for flow i, Λ_n is the arrival time of packet n and Γ_n is the size of packet n. The packets that become eligible, are then sent to a FIFO queue to be sent out through the egress interface. In this manner, the

maximum rate of the flow i is capped by μ_i.

Agents can change the service offered by Rate-Jitter regulators by adjusting the bandwidth share of each class. At every update of their value functions, they can change μ_i for each class to a value which they think will maximize the future returns.

4.3 QoS at Grid Resources

Similar to network QoS, we also implemented QoS through provisioning and reservation for GRs. For provisioning, we implemented a scheduler that prioritizes jobs by assigning weights, something that approximates GPS. The weight assigned to a job is dependent on its class. To provide reservation services, we implemented a mechanism in which each jobs get assigned a certain share of the CPU cycled, depending on its class.

4.3.1 CPU Provisioning

For provisioning at GRs, we implemented an algorithm that approximates Generalized Processor Sharing (GPS) ([62]). Jobs coming in to a server are put in a common queue. Each job is assigned a certain weight ϕ_i, which depends on its class. All jobs in the same class are assigned the same weight. The scheduler running at the GR processes all the jobs in parallel, and ensures that Equation 4.1 is adhered to at all times.

We have assumed that the cycles are perfectly divisible, and no atomic steps are present that require more than one cycle of the CPU time. If the total capacity of the CPU is C, then a job j with a weight ϕ_j receives the following amount of CPU capacity in the interval $[\tau, t]$. This assumes there are no new job arrivals and no job gets completed in that interval.

$$TS_j(\tau, t) = \frac{\phi_j C}{\sum_{k=1}^{K} \phi_k} \tag{4.6}$$

The capacity of a server is defined in terms of the number of *Millions of Instructions Per Second (MIPS)* it can execute.

The agent residing at the GR is able to modify the weights assigned to each job. This adjustment is done to try and maximize the expected return from jobs arriving in the future. Adjusting the weights of the jobs changes the share of CPU cycles that the job will receive from the scheduler.

4.3.2 CPU Reservation

Reservation of CPU cycles is implemented by defining the number of CPU cycles that each job will receive. This number of MIPS a job gets depends on which class it belongs to. The scheduler running on the GR ensures that each jobs receives the exact amount of MIPS that is due to it. If the amount of MIPS reserved by job j is M_j, then an admission control algorithm implemented at the server needs to make sure that the following condition is met

$$\sum_{j=1}^{N} M_j \leqslant C \tag{4.7}$$

where C is the capacity of the server in MIPS.

The agent running at the GRs can set the MIPS that each class should receive. By doing this, it can control how long each jobs takes to complete. The agent tries to make this adjustment in such a way that its returns from future jobs are maximized.

4.4 Using RL for Resource Allocation

We have implemented two algorithms based on Reinforcement Learning to adjust provisioning and reservation levels for routers and GRs - Watkins Q(λ) ([30] and SMART ([31]). These two algorithms have been introduced in Sections 2.2.1 and 2.2.2 respectively.

We have evaluated the performance of these two algorithms in both simulation and implementation on a testbed. In simulation, we can assume complete knowledge of the system state, but that is not possible with implementation. In order to account for this, some slight adjustments are needed in the RL algorithms for simulation and implementations. Sections 4.5 and 4.6 explain the design of the RL algorithms for simulation and implementation cases respectively.

4.5 Simulation

The simulations are discussed thoroughly in Chapter 5. The simulation consisted of two routers, two GRs, and two users sending their jobs to the GRs. The two users submitted different classes of jobs, therefore two classes of jobs existed in the system. The UBs responsible for each user were would decide which GR would process a submitted job. The UB would take the decision according to the agents value functions. Therefore the action taken at the agents was to decide which GR to use. The agents running at the network nodes would decide the service levels to be provided to each class, and the agents at the GRs would decide the resource allocation for each class. Therefore, the actions at the resources were to decide what service levels to use.

In order to formulate the resource provisioning or resource reservation problem as a reinforcement learning problem, we need to specify the state space, action space, reward structure and the update policies used by the agents.

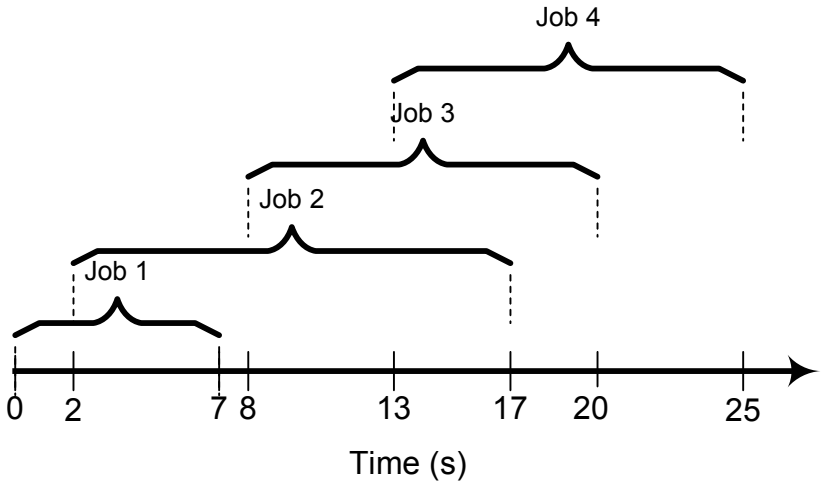

Figure 4.2: Sample Time-line Showing Generation and Completion of Jobs

4.5.1 State Space

For the state space of the UB, we use the response time of the job as the state dimension. The state space consists of all the possible values of the response time. Since the response time is continuous, the state space is also continuous. The response time of a resource is a good indicator of its load, since higher response times indicate high load on the system.

The UB is only interested in which GR can process its job the fastest. In our experiments, each user only sends one class of job, therefore the state for the UB is only composed of the response times of jobs for each GR. The RL agents maintains the state s_{UB} of the UB as

$$s_{UB} = \{\rho_{GR1}, \rho_{GR2}\} \tag{4.8}$$

where ρ_{GR1} is the response time of the first GR, and ρ_{GR2} is the response time of the second GR.

Figure 4.2 shows a sample time-line of job generation and completion for class k jobs. The current state of the system is always determined by the response time of the last job that was completed. For example, at time $t = 14$, the state dimension for that class is $\{\rho_{14,k}\} = \{7\}$, where as at time $t = 18$, the state dimension for

that class is $\{\rho_{18,k}\} = \{15\}$.

In the case of routers and GRs, each of them has one agent present. The state for these agents is simply the service levels being provided to each class of job. Since we have two classes of jobs, the state s_t at time t is

$$s_t = \phi_{1,t}, \phi_{2,t} \tag{4.9}$$

where $\phi_{i,t}$ is the service level for class 1 at time t and $\phi_{2,t}$ is the service level for class 2. The service level would be weights for WFQ on routers and GPS for GRs when resource provisioning is used. When resource reservation is tested, the service levels are determined by the bandwidth share for routers and CPU share for GRs.

4.5.2 Action Space

The action space defines the list of actions which the agents can choose.In case of a UB, the action space consists of choosing which GR should process the job. Since our simulation setup has two GRs, the action space consists of two items - 'Choose $GR1$' or 'Choose $GR2$'. 'Choose $GR1$' signifies that first GR has been chosen for processing, and 'Choose $GR2$' signifies that the second GR has been chosen.

For routers and GRs, an action signifies choosing the service level that will be used by the scheduler running at the resource. The list of service levels is predefined and depends on whether provisioning or reservation is being used by the schedulers. In the case of provisioning, a service level is the set of weights that will be used by the scheduler. These weights can be used for WFQ in network routers (Section 4.2.1) or for CPU Provisioning (Section 4.3.1). If provisioning is being used, a service level consists of defining what share of a resource will be assigned to job class. This is used to reserving bandwidth (Section 4.2.2) or

reserving CPU cycles (Section 4.3.2). Therefore the action space $\mathbb{A}$ consists of

$$\mathbb{A} = \{a_1, a_2, ..., a_n\} \tag{4.10}$$

where a is the action signifying choosing a service level, and n is the cardinality of the set of service levels. Since the list of actions is predefined and discrete, our action space is also discrete.

The actual list of actions used in experiments is defined in Sections 5.5, 5.6 and 5.7.

4.5.3 Reward Structure

Jobs which are successful (meet their deadline) are assigned a positive reward. A failure to meet the deadline is penalized by assigning a negative reward. Thus,if the deadline is met, the reward R is

$$R = 5 * \varsigma_i, \tag{4.11}$$

where ς_i is the scaling factor for class i. The scaling factor is used since some jobs may be valued more than others, and thus meeting their deadlines should be rewarded more. If a deadline is not met, the agent receives a negative reward R equal to

$$R = -5 * \varsigma_i \tag{4.12}$$

Some jobs have tighter deadlines than others. Thus, they may need higher network bandwidth and processing capacity. Therefore, if a job with a tighter deadline is successful, a larger reward is assigned to the agent. From an economic point of view, the cost of processing a job with a tight deadline should be higher than jobs which are not as sensitive to time. An optimal algorithm for resource allocation will maximize the revenue (reward) earned by the agent.

The rewards are generated by the UB when it receives a completed job back. The UB also rewards all the routers and grid resources that took part in processing this job, in a process detailed in Section 4.1.

4.5.4 Configuration of Reinforcement Learning Agents

In this section, we describe how the agents are configured at the UB, routers and GRs. We have already described the state and action spaces used by them, along with the reward policy in Sections 4.5.1, 4.5.2 and 4.5.3.

Configuration of Q-lambda algorithm

In one-step Q-Learning, we need the learning rate α, the discount rate γ, and the reward at each step for updating the Q-values for state action pairs, as shown in equation 2.4. Since we will be using eligibility traces to provide faster convergence, we need the trade-decay parameter λ shown in equation 2.5. In our experiments, λ is set to 0.2. λ determines how many past values of a state must be used to calculate the current value of the state. A value of 0 would mean only the previous step is used (TD(0)), and a value of 1 would mean all steps in the past are considered. A value of 0.2 means that only recent states are used to calculate the current value. This ensures that the agent does not rely on state values which are out-of-date.

The discount parameter γ is set to 0.9. A high value of γ means that the previous state values are decayed less. As discussed in Section 2.1.5, we also need to decay the exploration rate so that the Q-values are able to converge. The learning rates needs to be decayed the same reason. The learning rate and exploration rate are decayed according to the Darken-Cheng-Moody (DCM) search-then-converge procedure ([34]) each time a new action is chosen by the agent. The DCM procedure is shown in equation 4.13 and Figure 4.3 shows the effect of decaying the learning rate α using it. The effect on exploration rate ϵ is similar. A decaying

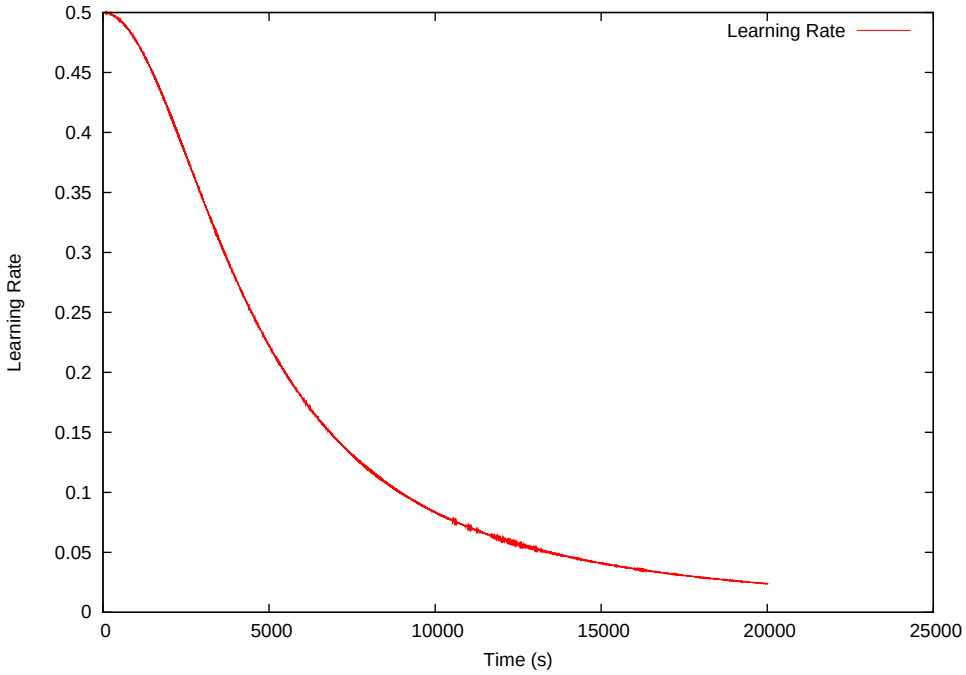

Figure 4.3: Effect of Darken-Chang-Moody Decay Algorithm on Learning Rate

exploration and learning rate help to ensure that the value functions stabilize over time. The initial learning rate α and the initial exploration rate ϵ are both set to 0.5.

$$(\alpha, \epsilon) = \frac{(\alpha, \epsilon)}{1 + \frac{n_s^2}{\varrho + n_s}} \tag{4.13}$$

where ϱ is the a constant and n_s is the number of times the CMAC has been updated. The decay constant ϱ is set to 5×10^6.

In order to reduce the storage requirements for the Q-values, we use CMAC tables for function approximation (Section 2.1.8). The CMAC needs a certain number of quantizing functions and resolution elements to run. Higher number of these will increase the memory requirements (Equation 2.3), will result in a lack of accuracy. We used 20 quantizing functions and 20 resolution elements for the CMAC.

Configuration of SMART algorithm

Section 2.2.2 shows how SMART can be implemented to solve a reinforcement learning problem. In order to calculate the average reward rate which is shown in equation 2.10, we need the learning rate and discount rate. The learning rate is set to 0.5 at the start of each experiment, while the discount rate is 0.9. The exploration rate is also set to 0.5 at the start of each experiment. The learning and exploration rates are decayed using the DCM search-then-converge procedure, similar to how they are decayed for the Q(λ) algorithm. In order to reduce storage needs for Q-values, we used the CMAC function approximator, as described in Section 2.1.8. The number of quantizing functions and resolution elements are set to 20.

At every decision epoch, when the value functions need to be updated, the average reward $\mathcal{AR}$ is updated as shown in equation 4.14.

$$\mathcal{AR} = (1 - \alpha) * \mathcal{AR} + \alpha \frac{t_{old} + \mathcal{AR} + R}{t_{new}} \tag{4.14}$$

where t_{old} is the time at the previous decision epoch, t_{new} is the current time, and R is the reward received during this epoch.

4.5.5 Update Policy

The update policy of each learner determines how a reward should be processed, how its value functions should be updated, and what action, if any, should be taken.

The update policy in Q(λ) and SMART requires the action for which the value functions are being updated, the state previous to the action being taken, the state after the action was taken, and the reward that the system received for taking the action. In addition to these, SMART also requires the the sojourn time between this action being taken and the next decision epoch. Since the learning

policy requires the the state at which the action was taken, we need to keep track of the state at which a job was submitted by the system. In our experiments, we kept this information inside the submitted job itself. However, in an actual implementation, the UB can manage all this information by keeping track of the GR load information from the Grid Information System (GIS), and state of routers and network links with the help of a Network Management System (NMS) using Simple Network Management Protocol (SNMP).

Once a job is received at the UB, the completion time of the job is calculated and a reward is generated according to the reward policy in Section 4.5.3. The current state of the system are also calculated. The state of the system when the job was submitted, which is stored in the job itself, is also extracted. All this information is then supplied to the agent running at the UB.

When the agent receives a reward, it is used to calculate the TD error according to Equation 2.6. The TD error is used to update the CMAC function approximators. This is common for agents running Q(λ) and SMART. The same process takes place at the agents running on routers and GRs.

4.6 Implementation on Testbed

For the implementation, we have evaluated the performance of Watkins Q(λ) learner and the SMART algorithm. The CPU Resource Manager can assign weights to jobs, thereby providing facilities for CPU provisioning. The Network Resource Manager supports reservation of bandwidth by jobs, and the amount of reservations can be adjusted by a learning agent running at the each router.

The action space, state space, update policies etc. were similar to that for simulation. The only thing that was different from the simulation scenario described

in Section 4.5 was the reward structure.

$$R_i = \begin{cases} 10 & \text{if successful} \\ \frac{-5}{\rho_i - d_i} & \text{if failure} \end{cases} \tag{4.15}$$

where R_i is the reward for job i, ρ_i is the total processing time of the job (response time) and d_i is the deadline for the job. The penalty is structured in the way shown in equation 4.15 so that jobs which miss their deadline by larger margins generate a larger penalty.

We evaluated the performance of our reinforcement learning based methods described above in simulation and implementation on a testbed. Chapter 5 describes in detail our simulation setup and results. and also deals with implementation details and results.

Chapter 5

Performance Evaluation

In this chapter, we discuss the details of the simulations we carried out, and the details on implementation and experiments done on a testbed. Sections 5.1 to 5.8 describe the simulation setup, the various experiments carried out, and the results obtained from them. Sections 5.9 to 5.13 describe the implementation on a testbed, and the results obtained from the experiments carried out on it.

5.1 Simulation

In this chapter, we discuss the details of the simulations we carried out to test our Reinforcement Learning QoS framework. All the tests are carried out using the GridSim simulation tool. We have tested both provisioning and reservation, using the methods described in Chapter 4.

5.2 Simulation Scenarios

We have simulated three different scenarios in our experiments -

- Scenario I - In the first scenario, agents run only on the brokers attached to each user (UB1 and UB2). The broker is a proxy for the user, and it decides which Grid Resource (GR) should process a job. In our experiment

the brokers have a choice between two resources (GR1 and GR2). The agents try to minimize the response time for the user they are attached to by trying to learn the optimum scheduling policy. The agents measure completion times of jobs and reward or penalize themselves depending on the response time of the submitted jobs.

- Scenario II - In the second scenario, agents run on the network routers and GRs. Here the agents have complete control over the resource provisioning or reservation. At the end of each job, the UBs calculate the reward or penalty that the routers and GRs should receive, and sends this data to the agents running on them. The agents at routers and GRs then update their resource allocation policies, and apply the new policy to their schedulers. There is no agent running on the UB in this scenario.

- Scenario III - In the third scenario, we explore providing combined QoS on UBs, routers and GRs. Agents are enabled on the UBs as well as the routers and GRs. On the completion of a job, the UBs generate a reward or penalty signal. The UBs use this signal to update the agents running on themselves, as well as the agents at routers and GRs.

5.3 Benchmarking

The agents running at the UBs, routers and GRs are run in different configurations to evaluate the performance of the Q(λ) and SMART algorithms. The agent at the UB is run in four different configurations, while the agents at routers and GRs were run in three different configurations.

5.3.1 Configurations of Agents at UBs

The UB is run in the following configurations -

- RR - In the first case, the UB runs in Round Robin (RR) mode. In RR mode, jobs are sent alternatively to GR1 and GR2. Therefore the jobs are distributed evenly among the GRs, irrespective of their response times.

- ExpAvg - In the second case we use exponential averaging. At the completion of each job, we calculate its response time, and use that to update an exponentially weighted average for each resource. A new job is always sent to the resource with lower average response time. Equation 5.1 shows the formula used to calculate the average response time.

$$EA_{it} = \alpha * \rho_{it} + (1 - \alpha) * EA_{i(t-1)} \tag{5.1}$$

  where EA_{it} is the exponentially weighted average of resource i at time t, ρ_{it} is the response time of the last job at resource i and α is the learning rate, which is set to $\alpha = 0.5$.

- QL - In the third case, the agents use Watkin's Q-Learning algorithm. In this case, the greedy action is chosen with a probability of $1 - \epsilon$, and an exploratory action is chosen the rest of the time. ϵ starts with a value of 0.5, but it is decayed using the Darken-Chang-Moody Search-then-Converge technique as shown in Section 4.5.4. The other parameters used in Q(λ) are also provided in the same section.

- SMART - In the fourth case, the agents use the SMART algorithm. Like the QL case, the agents explore their state-space with a probability of ϵ, while taking the greedy action the other times. The complete details are provided in Section 4.5.4.

5.3.2 Configuration of Agents at Routers and GRs

The agents at routers and GRs are also run with three different cases.

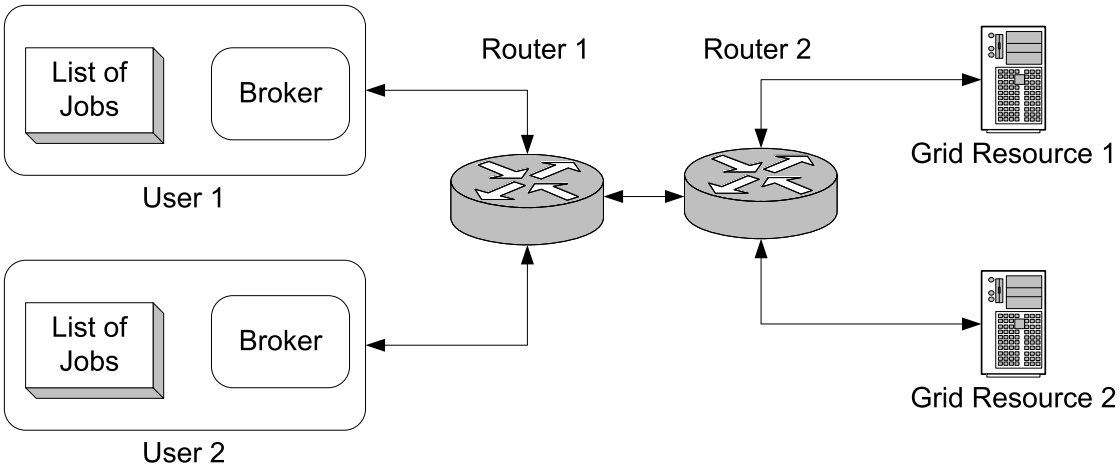

Figure 5.1: Simulation Setup

- Static - Here, the resource allocations for each class of jobs are pre-configured, and they do not change throughout the course of the experiment.

- QL - In this case, the agents at routers and GRs use the Q(λ) algorithm. Each time a reward or penalty is received, the agent updates its value functions, and chooses a new resource allocation to be used by the resource. It takes the greedy action with a probability of $1-\epsilon$, and chooses an exploratory action the other times. The learning rate and exploration rate are set to 0.5 at the start of the experiment, but decays as described in Section 4.5.4.

- SMART - The agents at routers and GRs use the SMART algorithm to decide the resource allocation. Similar to the QL case, exploratory actions are taken with a probability of ϵ, with the non-exploratory actions being taken otherwise. The decay in learning rates and exploration rates are the same as the QL case.

5.4 Simulation Setup

5.4.1 Topology and Characteristics

Figure 5.1 shows the topology that was simulated in GridSim. The scenario consists of two users of the grid, with two grid resources and two routers in between the users and resources. All the network links had a bandwidth of 1 Megabit per second (Mbps)and a propagation delay of 5 ms. The MTU for each packet in the network is 1000 bytes. The routers are able to process data at 1 Mbps. GR1 has a processing capacity of 250 Million Instructions Per Second (MIPS), while GR2 has a processing capacity of 350 MIPS.

GR1 does not support providing differentiated services. It gives equal preference to both the classes, by dividing its cycles equally among both classes of jobs. On the other hand, GR2 does support differentiated service, and the service it provides to a job depends on its class.

Table 5.1: Characteristics of Jobs in Simulation Setup

Class	Job Size (MIPS)	Data Size (bytes)	Mean Generation Delay (s)	Deadline (s)
1	150	50000	2.5	4
2	300	100000	2.5	15

Table 5.1 shows the configuration of the jobs of the two classes that were sent to the GRs. Class 2 jobs require more processing power and network bandwidth, but Class 1 jobs require a lower response time. Class 2 jobs are designed to model bulk or background jobs. Class 1 jobs are modeled as jobs requiring immediate attention and fast response times, therefore the scaling factor for Class 1 and 2 were 1.5 and 1 respectively. This means that completing a Class 1 job within its deadline generated a reward that was 1.5 time higher than Class 2 jobs. Failing to meet the deadline for a Class 1 job generates a penalty that is 1.5 times the penalty generated when a Class 2 job fails. User 1 always sends Class 1 jobs,

while User 2 sends jobs of Class type 2. The delay between sending out two jobs is determined by an exponential distribution [65].

Each simulation is run for 25,000 seconds of the simulation clock.

5.5 Scenario I - User Level Job Scheduler

In this section, the use of a user level scheduler is studied. A user may have access to various grid resources, with some providing him better service than others, depending on the load condition. Some grid or network resources may support providing Quality of Support (QoS), while other resources may be set to static configurations. It is the function of the UB to select the resource from which it expects to receive the best service.

In this scenario, the routers and GRs do not provide any QoS, therefore they are set at static service levels. GR2 gives a higher preference to Class 1 jobs. It can do this either by reserving more cycles for Class 1 and using a CPU share algorithm (Section 4.3.2), or by assigning higher weights to Class 1 jobs (Section 4.3.1). We conducted the experiment with both the above alternatives, and the results are given in Sections 5.5.1 and 5.5.2.

5.5.1 Using Reservation on GRs

In this setup, GR1 divides its CPU cycles equally among both classes. GR2 reserves 65% of its cycles for Class 1 jobs, and 35% for Class 2. The experiment is run with the four cases listed in Section 5.3.1. In this scenario, we are interested in studying the processing time of the jobs at the GRs, since the service at the routers is the same for both classes of jobs. Therefore, we only list the processing time at the GRs, rather than the response time.

Results are provided in Table 5.2, and graphed in Figures 5.2 and 5.3. It can be seen that there is not much observable difference in the performance of RR,

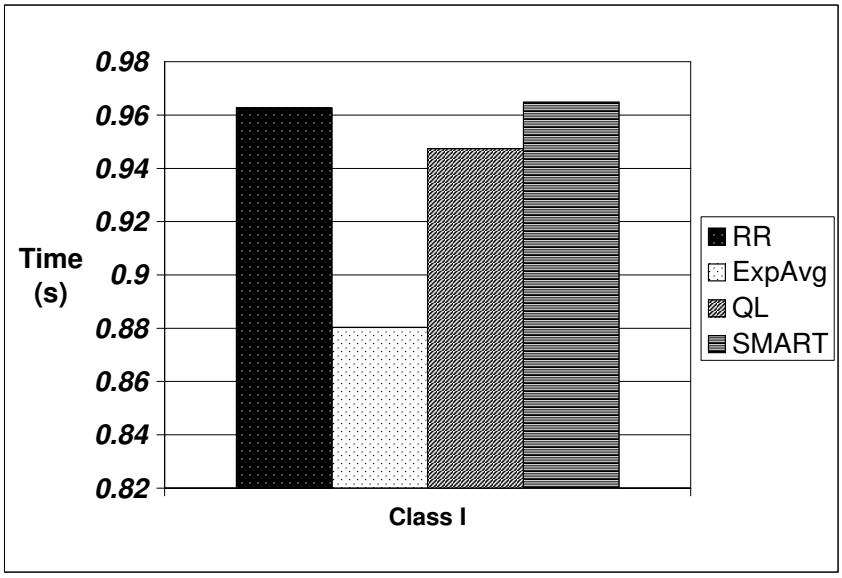

Figure 5.2: Average Processing Time (s) in Scenario I with Reservation (Class 1)

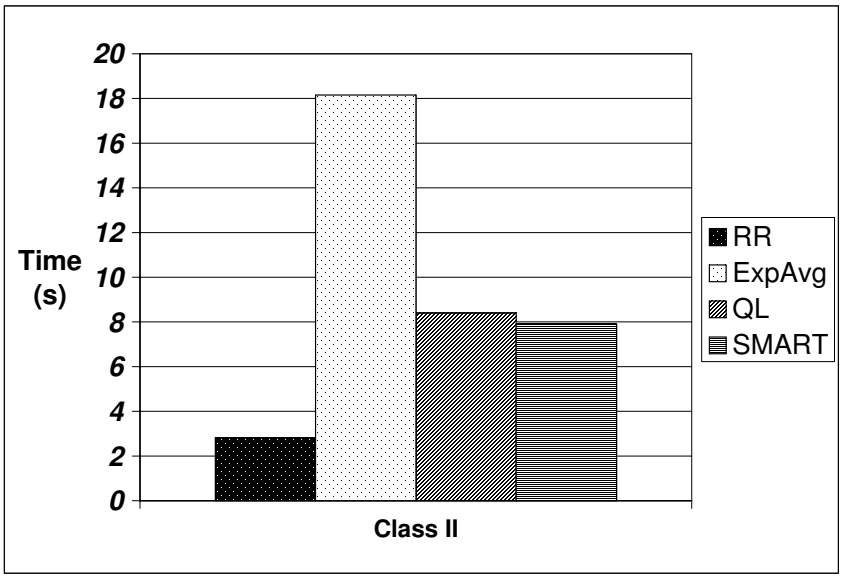

Figure 5.3: Average Processing Time (s) in Scenario I with Reservation (Class 2)

Table 5.2: Average Processing Time for Jobs in Scenario I with Reservation

Class	RR (s)	ExpAvg (s)	QL (s)	SMART (s)
1	0.962621	0.880351	0.947469	0.964863
2	2.815509	18.158761	8.401099	7.913707

QL and SMART for Class 1 jobs. ExpAvg does better for Class 1 than the other three, but the average processing time for Class 2 jobs is worse than the deadline for Class 2 (15 s). This can be understood in the following way - GR1, which can execute 200 MIPS provides equal service to Class 1 and Class 2 jobs. GR2, which can execute 350 MIPS, reserves 65% of its CPU cycles for Class 1, which means that Class 1 jobs get 227.5 MIPS at GR2, and Class 2 jobs get 122.5 MIPS. The learning agent at UB1 observes that GR2 gives it better service. Therefore, the agent at UB1 running QL or SMART algorithm tends to send most of it jobs to GR1. On the other hand, UB2 observes it is getting 125 MIPS from GR1 and 122.5 MIPS from GR2, so the agent there sends almost equally to both GRs. This can be observed in Figure 5.4. Due to almost all jobs from Class 1 being sent to GR2, its average processing time is slightly higher than the RR case, though still well within the deadline. When the RR scheme is used, both classes of jobs are sent evenly to both GRs. Thus, a learning agent deployed only at the UBs does not perform any better or worse than a simple RR algorithm.

The ExpAvg algorithm does obtain a average lower processing time for Class 1, but its average processing time for Class 2 is even higher than the deadline for the response time. The reason for this can be seen in Figure 5.5. It can be observed that when the curve for jobs of any class being sent to GR1 is rising, the corresponding curve for jobs of that class being sent to GR2 is flat, and vice-versa. This is because ExpAvg sends all its jobs to the resource with the lower response time. For example, if UB1 observes a lower response time average from GR1, it will send all its jobs to GR1. GR1 then gets swamped with Class 1 jobs, and starts to perform poorly. However, since the average response time is weighted,

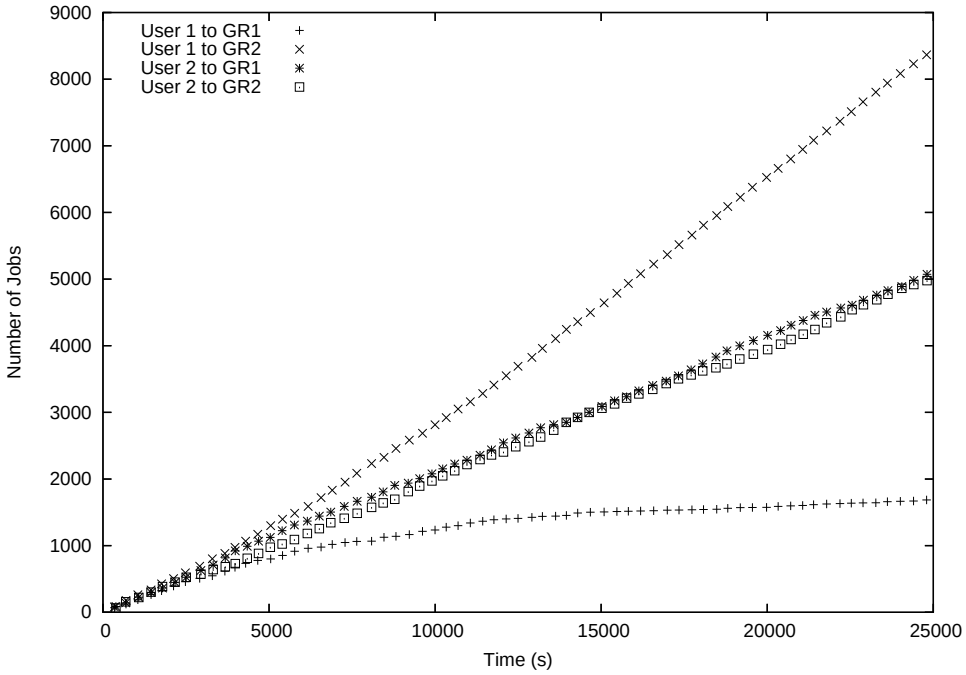

Figure 5.4: Distribution of Jobs in Scenario I using Reservation (QL)

UB1 keeps sending jobs to GR1 until its average response time climbs above GR2. UB1 then starts to send all its jobs to GR2, and this cycle continues. Thus a simple weighted averaging learning technique is insufficient to provide good QoS, when resource reservation facilities are provided.

5.5.2 Using Provisioning on GRs

In this setup, GR1 assigns equal weights to both jobs of classes. GR2 assigns Class 1 jobs thrice the weight of Class 2 (3:1) jobs. The scheduler running at both GR1 and GR2 is SCFQ. Similar to Section 5.5.1, the experiment is run with the four cases listed in Section 5.3. Results are given in Table 5.3.

In the case of provisioning, the service received by a class of jobs is not guaranteed. The number of CPU cycles that a Class 1 job gets depends not only on how many other Class 1 jobs are present, but also on how many Class 2 jobs are

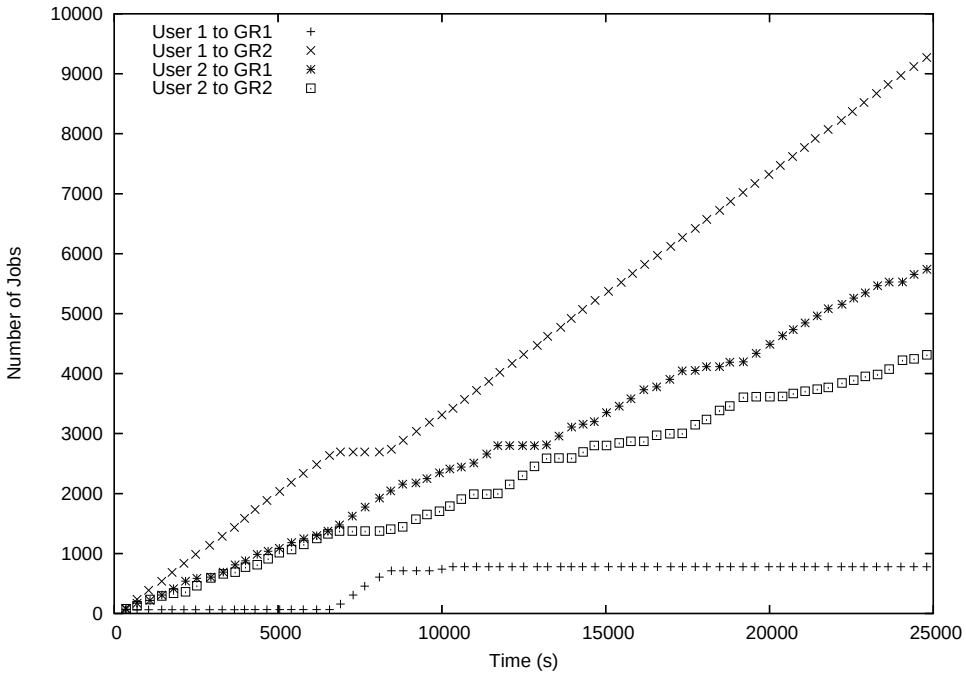

Figure 5.5: Distribution of Jobs in Scenario I using Reservation (ExpAvg)

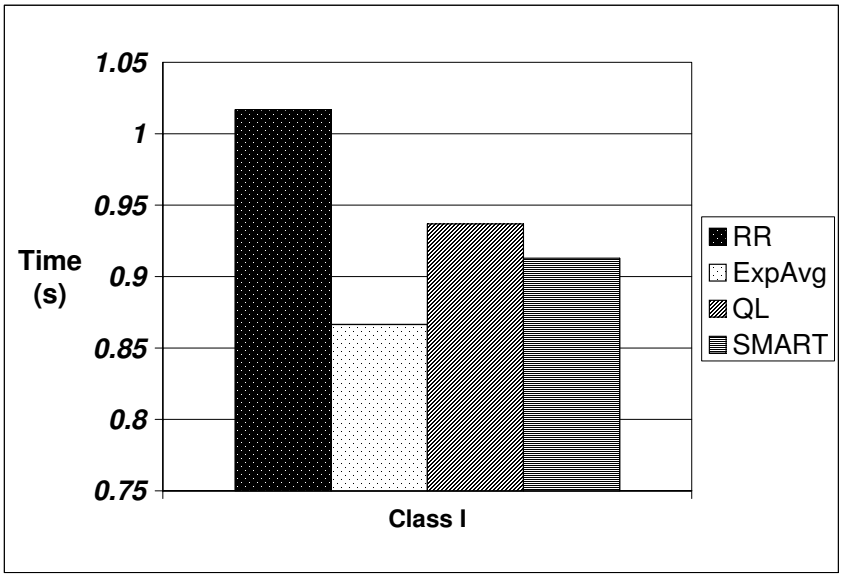

Figure 5.6: Average Processing Time (s) in Scenario I with Provisioning (Class 1)

Table 5.3: Average Processing Time for Jobs in Scenario I with Provisioning

Class	RR (s)	ExpAvg (s)	QL (s)	SMART (s)
1	1.016818	0.866499	0.936889	0.912874
2	1.387380	1.484839	1.653235	1.596051

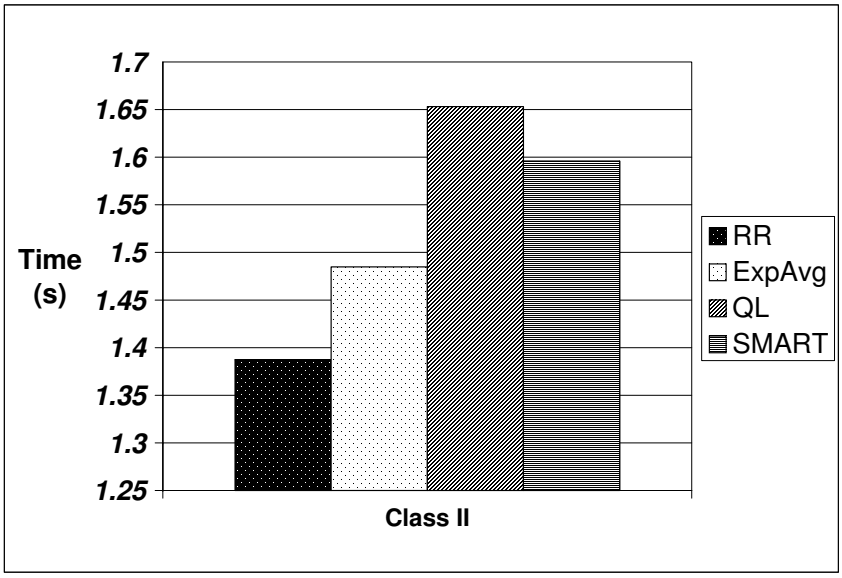

Figure 5.7: Average Processing Time (s) in Scenario I with Provisioning (Class 2)

present in the system simultaneously. Therefore in this case, the agents running QL and SMART are able to obtain a lower processing time than an agent distributing jobs in a RR manner. When using RR, both both resources receive equal amount of jobs, even though GR1 has only about 71.4% the capacity of GR2. The agents running learning algorithms like QL or SMART distribute jobs to servers where they receive better service, as can be seen in Figure 5.8. It can be seen that UB1 sends most of its jobs to GR2. However, provisioning does not insulate the performance for Class 1 jobs completely from other jobs. UB2 also receives a low response time from GR2 since it has more capacity, and thus sends most of its jobs to GR2.

ExpAvg is able to obtain the best response times when provisioning is used

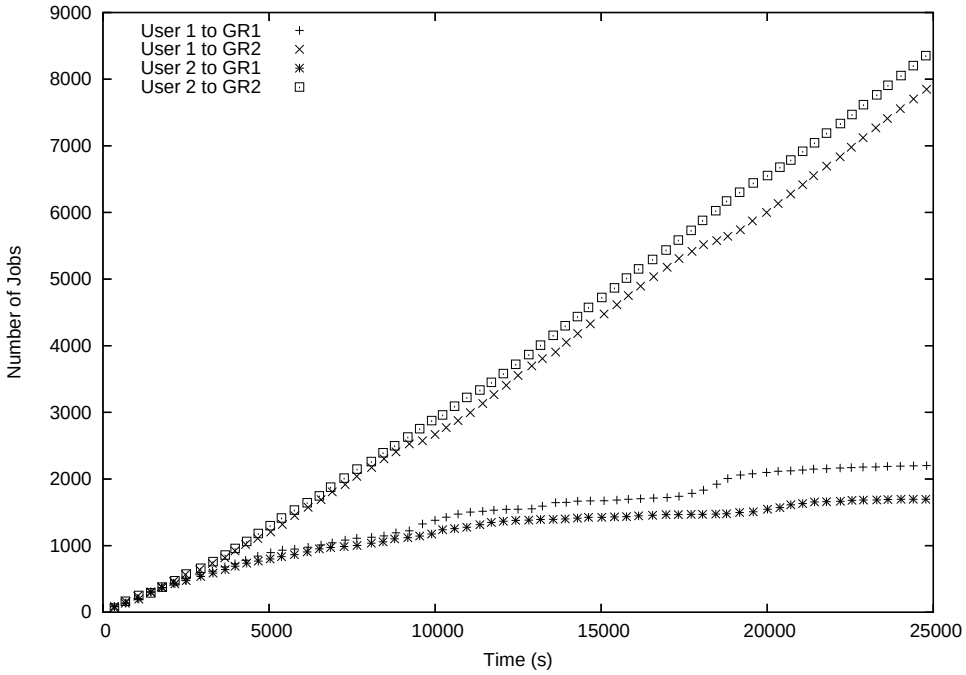

Figure 5.8: Distribution of Jobs in Scenario I with Provisioning (SMART)

at the resources. Once again, this is because having GR2 does not provide a guaranteed service level to Class 1. As a results, Class 2 jobs are also able to get good service at GR2, even when Class 1 jobs are present. From Figure 5.9, it can be seen that both UB1 and UB2 send most of their jobs to GR2.

5.6 Scenario II - Resource-Level RL Management

In the second scenario, agents are deployed only on routers and GRs. The UBs submit jobs in a Round Robin fashion to the two resources, therefore each router receives an equal amount of jobs of each class. We studied the effect of providing QoS on routers and GRs through reservation and provisioning. The results are discussed in Sections 5.6.1 and 5.6.2 respectively.

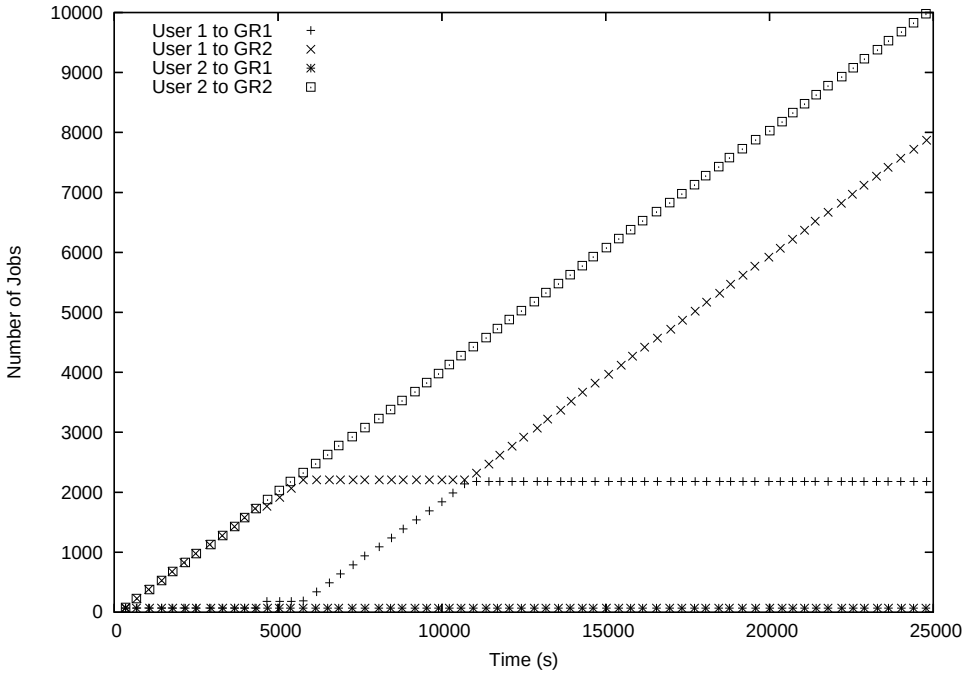

Figure 5.9: Distribution of Jobs in Scenario I with Provisioning (ExpAvg)

5.6.1 Using Resource Reservation on Routers and GRs

At both the routers, banwidth reservation is provided by using a rate-jitter scheduler (Section 4.2.2). When the agent at routers is running the QL or SMART algorithm, it can choose the reservation levels to use. The agent can choose one of the reservation level pair from following list - {(40%,60%), (50%,50%), (60%,40%), (70%,30%)}, where the first element of the set represents the percentage of CPU cycles reserved for Class 1, and the second element represents the percentage of CPU cycles reserved for Class 2.

Agents are also present at GRs, and when running the QL or SMART algorithm, they can choose the reservation levels to use for each class of job. Similar to the routers, the reservation levels that the agent can choose are as follows - {(40%,60%), (50%,50%), (60%,40%), (70%,30%)}. The CPU cycles reserved for a class of jobs is shared equally among all jobs of that class running simultaneously

on that GR.

The agents at routers and GRs are run in three different configurations : Static, QL and SMART, as described in Section 5.3.2.

Table 5.4: Average Response Time for Jobs in Scenario II with Reservation

Class	Static (s)	QL (s)	SMART (s)
1	2.677932	2.338313	2.335429
2	5.630762	4.728753	4.742591

Table 5.5: Average Processing Time for Jobs in Scenario II with Reservation

Class	Static (s)	QL (s)	SMART (s)
1	0.962621	0.644755	0.644895
2	2.815509	1.803996	1.810912

It can be seen from Tables 5.4 and 5.5 that the agents running QL and SMART are able to lower the response and processing times for Class 1 jobs. With the efficient reservation of resources, we are also able to lower response and processing times for Class 2. In the static configuration, GR1 had reserved 50% of its CPU cycles for Class 1, and GR2 had reserved 65% of its cycles for Class 1. The routers also had reserved 50% of their bandwidth for Class 1. Thus, Class 1 was receiving a much larger share of CPU resources, and an equal amount of network bandwidth, even though it only required half the processing power and network bandwidth as compared to Class 2. The QL and SMART are able to correct this overprovisioning of resources, and thus obtain better QoS for Class 2 also.

5.6.2 Using Resource Provisioning on Routers and GRs

In this section, we changed the schedulers on the routers and GRs, so that they perform resource provisioning rather than resource reservation. Similar to the previous section, the agents running at the routers can decide the weights for different classes of jobs when they are utilizing the QL or SMART algorithm. The

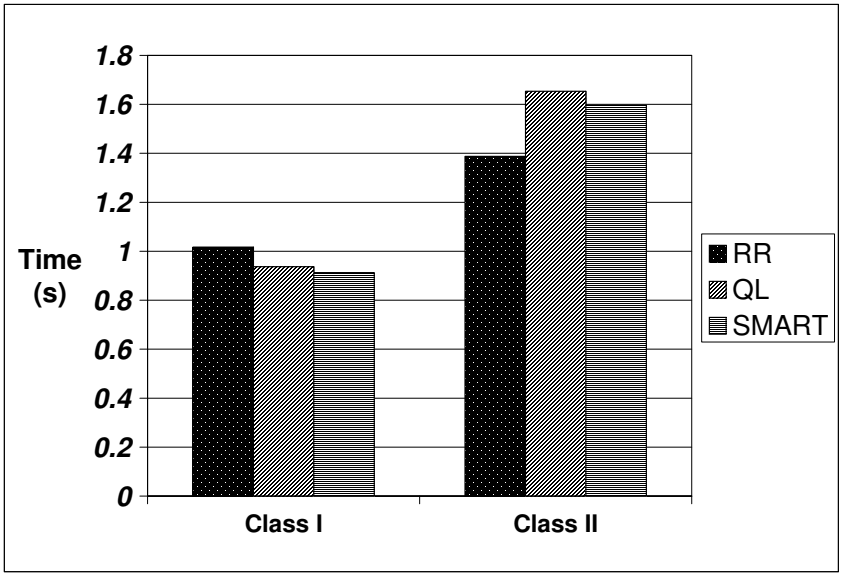

Figure 5.10: Average Response Time (s) in Scenario II with Reservation

scheduler running on the agents processes the incoming packets according to the SCFQ algorithm (Section 4.2.1). The agents at routers can choose one of the following weight sets - {(0.5:1), (1:1), (2.5:1), (5:1) and (7.5:1)}, with the first element being the weight of Class 1 jobs, and the second element being the weight of Class 2 jobs. The agents running at GRs have the same set of weights to choose from. The schedulers at the GRs use the GPS algorithm to provision CPU cycles to users (Section 4.3.1).

When running in static mode, the routers and GR1 are setup to provide 1:1 provisioning for Class 1 and 2. GR2 is configured to assign thrice the weight to Class 1 jobs as compared to Class 2.

Table 5.6: Average Response Time for Jobs in Scenario II with Provisioning

Class	Static (s)	QL (s)	SMART (s)
1	2.664450	3.456912	3.438866
2	4.216304	3.864805	3.869627

Table 5.7: Average Processing Time for Jobs in Scenario II with Provisioning

Class	Static (s)	QL (s)	SMART (s)
1	1.016818	1.047533	1.037764
2	1.387380	1.384302	1.388785

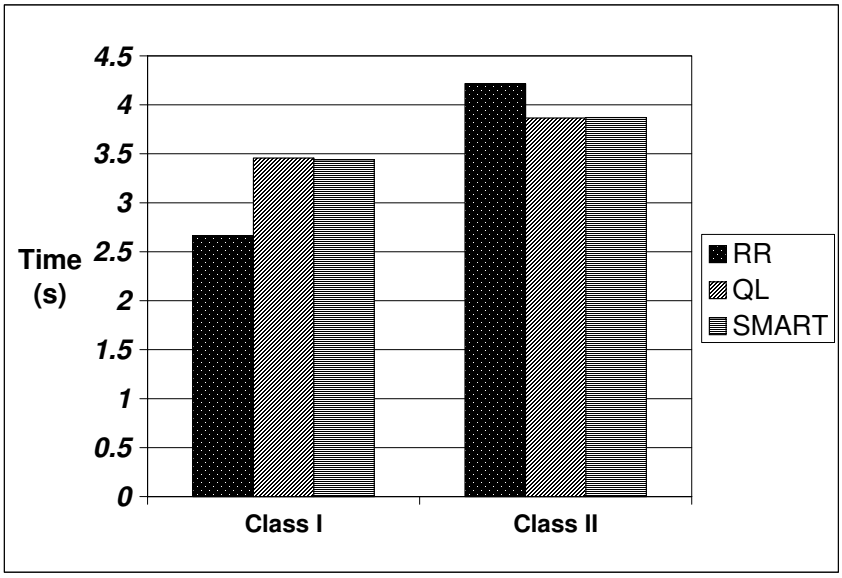

Figure 5.11: Average Response Time (s) in Scenario II with Provisioning

Tables 5.6 and 5.7 summarize the average response time and the processing time obtained by using the three schemes, with provisioning enabled on routers and GRs. It can be seen that the response time and processing times for Class 1 are higher than when reservation is used (Section 5.6.1 - Tables 5.4 and 5.5.). This is again because higher provisioning levels do not guarantee performance. Similar to Section 5.5.2, Class 1 jobs experience an increase in response times when Class 2 jobs also queue up. This also leads to Class 2 getting lower response times when provisioning is used rather than reservation. This can be observed by comparing Table 5.6 (Provisioning Case) to Table 5.4 (Reservation Case).

5.7 Scenario III - Integrated QoS

In Sections 5.5 and 5.6 we saw the effect of running agents using Reinforcement Learning to perform resource reservation or provisioning. The agents were either run on the UBs, or they were run on resources like routers and GRs. Using RL, they are able to learn a policy that can benefit classes that require quick response time, without overly penalizing other classes of jobs. In this section, we study the effect of performing resource reservation or provisioning along with scheduling by the UBs, with all the actions being decided by Reinforcement Learning policies.

Like Sections 5.5 and 5.6, agents at resources are responsible for resource allocation, and take actions which they think will maximize the return of jobs in the future. Agents at UB schedule the jobs according what they feel is the best policy to receive low response times. The same algorithm is used at UBs and resources. Thus, when the UBs use QL to learn the optimum scheduling policy, agents at routers use QL to learn the best resource allocation policy.

In order to compare the performance of agents utilizing Reinforcement Learning, we also ran the UB in RR and ExpAvg mode (Section 5.3.1). When the UBs run in RR or ExpAvg mode, the agents at routers and GRs run in static mode.

5.7.1 Using Resource Reservation on Routers and GRs

In this part of the experiment, the agents running on routers and GRs were responsible for deciding how much of the resources would be reserved for each class of jobs. The list of actions available to the agents at routers and GRs is the same as Section 5.6.1.

When the agents at routers and GRs were run in static mode, the network bandwidth and CPU cycles at GR1 is shared 1:1 among Class 1 and 2. GR2 reserves 65% of its bandwidth for Class 1, and the rest is reserved for Class 35%.

It can be seen from Tables 5.8 and 5.9 that the response times for Class 1 and

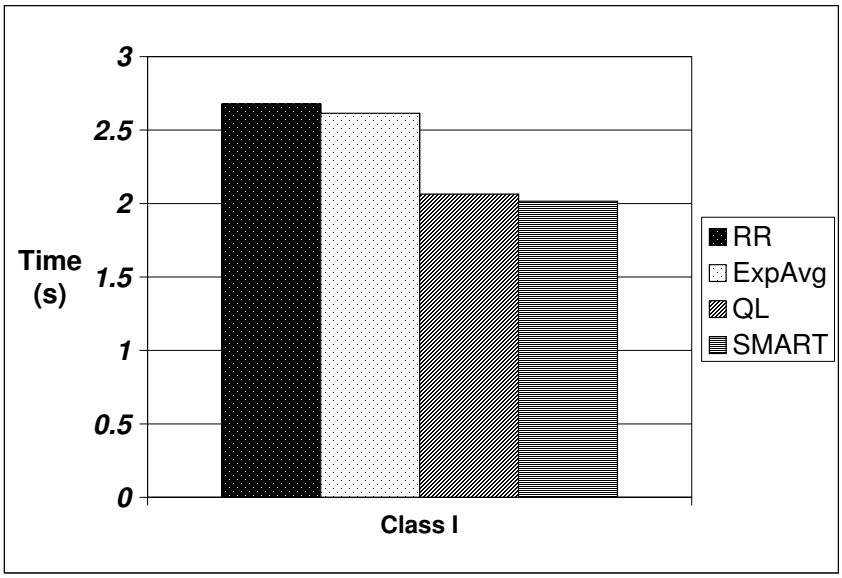

Figure 5.12: Average Response Time (s) in Scenario III with Reservation (Class 1)

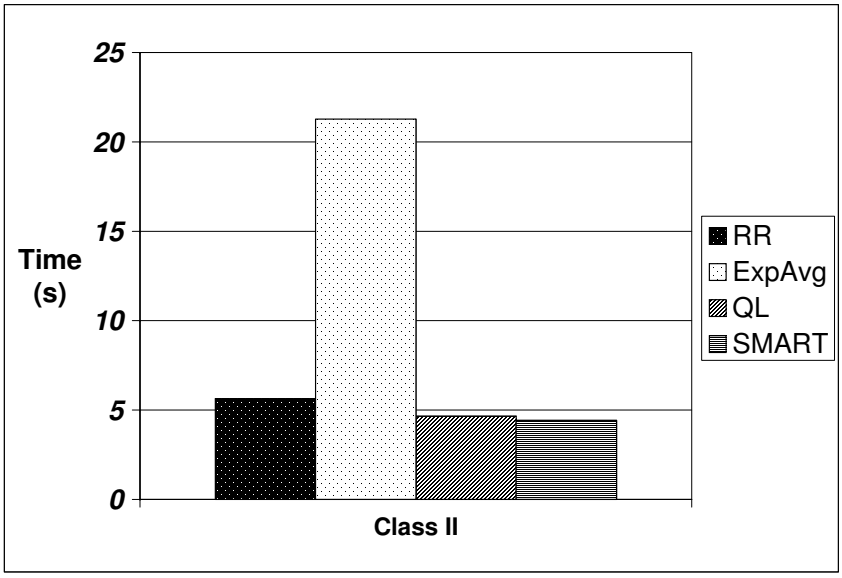

Figure 5.13: Average Response Time (s) in Scenario III with Reservation (Class 2)

Table 5.8: Average Response Time for Jobs in Scenario III with Reservation

Class	RR (s)	ExpAvg (s)	QL (s)	SMART (s)
1	2.677932	2.614117	2.064217	2.016163
2	5.630762	21.276755	4.651910	4.423825

Table 5.9: Average Processing Time for Jobs in Scenario III with Reservation

Class	RR (s)	ExpAvg (s)	QL (s)	SMART (s)
1	0.962621	0.880351	0.307630	0.257287
2	2.815509	18.158761	1.742260	1.550361

2 are better when the agents at UBs, routers and GRs use the QL or SMART algorithm, as compared to running the agents in static, RR or the ExpAvg mode. The response times for Class 1 and Class 2 is better than what is obtained by using resource reservation only at routers and GRs (Section 5.6.1). The processing times are also better than what is obtained with only scheduling enabled at UBs (Section 5.5.1). The response times for class 2 are beyond the deadline for the same reasons as explained in section 5.5.1.

Figure 5.14 shows how the jobs are distributed when agents use QL at UBs to learn scheduling policy, and agents use QL at routers and GRs to learn resource allocation policies. Both UB1 and UB2 send less jobs to GR1, which has a lower processing capacity.

5.7.2 Using Resource Provisioning on Routers and GRs

In this section, we compare the performance of using QL or SMART to perform resource provisioning at routers and GRs while UBs also simultaneously use the two algorithms to determine their scheduling policies. The list of actions available to the agents at routers and GRs is the same as Section 5.6.2. Similar to Section 5.7.1, in order to benchmark the performance of these algorithms, we used RR and ExpAvg scheduling policies at UBs, and static resource allocation levels at routers and GRs. When running in static mode, schedulers at routers and GR1

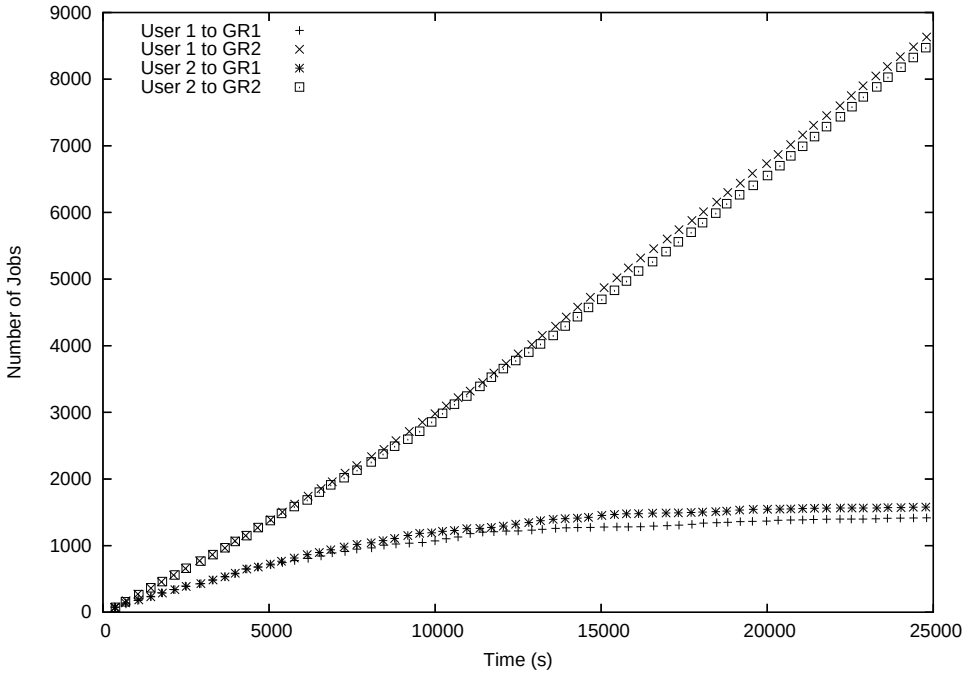

Figure 5.14: Distribution of Jobs in Scenario III using Reservation (QL)

were configured to give equal weightage to Class 1 and Class 2 jobs. GR2 was configured to give Class 1 jobs thrice the weight of Class 2 jobs.

Table 5.10: Average Response Time for Jobs in Scenario III with Provisioning

Class	RR (s)	ExpAvg (s)	QL (s)	SMART (s)
1	2.664450	2.500774	3.368479	3.141526
2	4.216304	4.226257	3.888730	3.919203

The results of using RL to decide resource allocation and scheduling policies are given in Tables 5.10 and 5.11. The response time for Class 1 and Class 2 jobs are comparable to those obtained by RR and ExpAvg schemes, but not better. The response times for both classes are slightly better than what is obtained by running the agents in RL mode on routers and GRs only (Section 5.6.2 - Table 5.6). The processing times for Class 1 jobs are similar to those obtained by running the agent in RL mode the the UB also, but the processing time for Class 2 is better

Table 5.11: Average Processing Time for Jobs in Scenario III with Provisioning

Class	RR (s)	ExpAvg (s)	QL (s)	SMART (s)
1	1.016818	0.866499	1.065486	0.934184
2	1.387380	1.484839	1.448755	1.482444

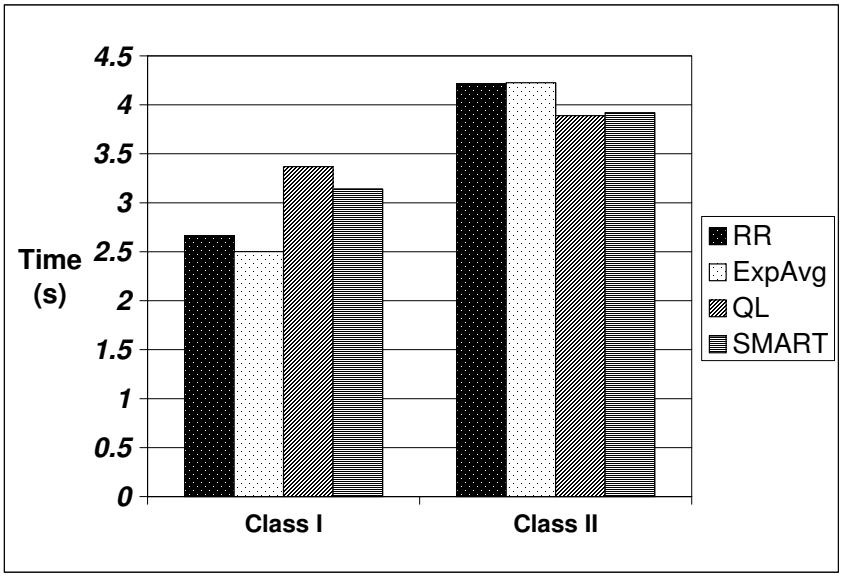

Figure 5.15: Average Response Time (s) in Scenario III with Provisioning

(Section 5.5.2 - Table 5.3).

Figure 5.16 shows how the jobs are scheduled by the UBs when they are using the SMART algorithm for scheduling. Compared to Figure 5.14, more jobs are sent to GR1, which is a weaker resource than GR2, leading to worse response and processing times as compared to Section 5.7.1.

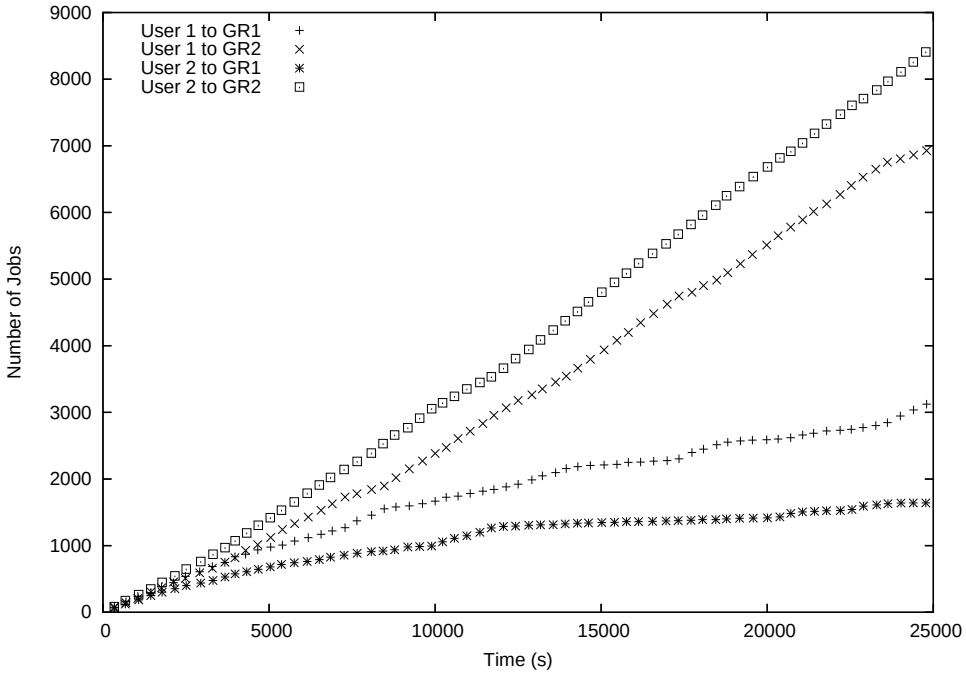

Figure 5.16: Distribution of Jobs in Scenario III using Provisioning (SMART)

5.8 Discussion

5.8.1 Reservation vs. Provisioning

From our experiments, it can can be seen that reservation of resources is able to obtain lower response and processing times for Class 1 compared to provisioning. The reason for this is that reservation is able to guarantee a certain level of service to jobs. When GR2 is reserves 65% of its cycles for Class 1, incoming Class 1 jobs share that evenly, irrespective of how many Class 2 jobs are also queued at the system. Thus, QoS differentiation is more distinct in the case of reservation as compared to the case of provisioning.

The drawback with reservation is that it can lead to wastage of resources. When no jobs of a class exist, its reserved cycles will be wasted. This problem can be overcome if the unused cycles are shared among other jobs, but could be taken away as soon as a job of a reserved class type arrives at the system. Another

alternative is to use a learning system like the ones discussed in this thesis, which can analyze the pattern of incoming jobs and try to minimize resource wastage.

5.8.2 Q-Learning vs. SMART

In all our experiments, we tested the performance of two RL algorithms. Q(λ) is used when the updates to the value functions take place at each unit of time. Thus it requires regular updates from the system. In a Grid environment, where jobs could be of varying duration, it might be difficult to perform such updates on a regular basis. The SMART algorithm is modeled for SMDP problems, where the sojourn time between updates can be of variable length. Due to the varying run-length of jobs in grids, it would be preferable to have a learning system based on the SMART algorithm.

5.8.3 Policy Learnt by User Brokers

Figure 5.17 shows the value function and the policy learnt by the QL algorithm at the UBs in Scenario III. The graph shown is for the case when reservation is used to perform resource allocation (Section 5.7.1). The y-axis represents the numerical reward that each action has accumulated over the period of the experiment. It can be seen that at the end of the simulation, the algorithm running at UB1 prefers sending jobs to GR2 as compared to GR1. When response times are lower than 10 seconds, UB2 gives an almost equal preference to both jobs. This is what leads to the job distribution seen in Figure 5.14.

The value functions have a lot of variance for high response time situations. This is because the average response time is much lower, as is seen in Table 5.8. Therefore situations where the response time is greater than 10 seconds are rare. Consequently, the state-action space at those levels are not explored extensively, leading to the value functions not stabilizing for high values of response times.

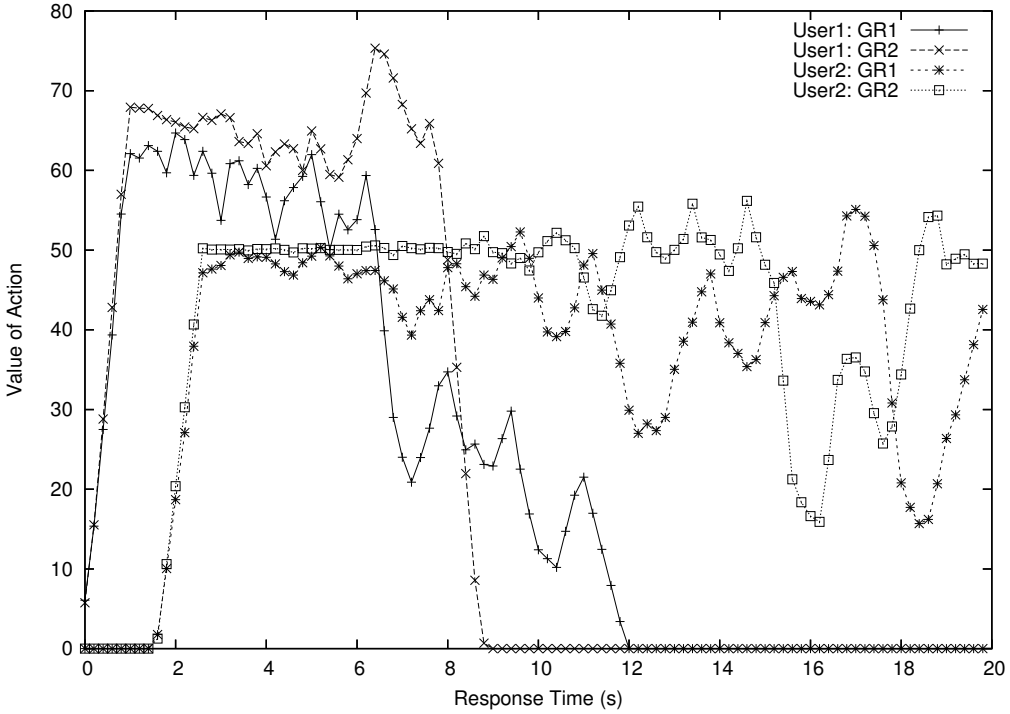

Figure 5.17: Value of Choosing GR1 or GR2 for User1 and User2

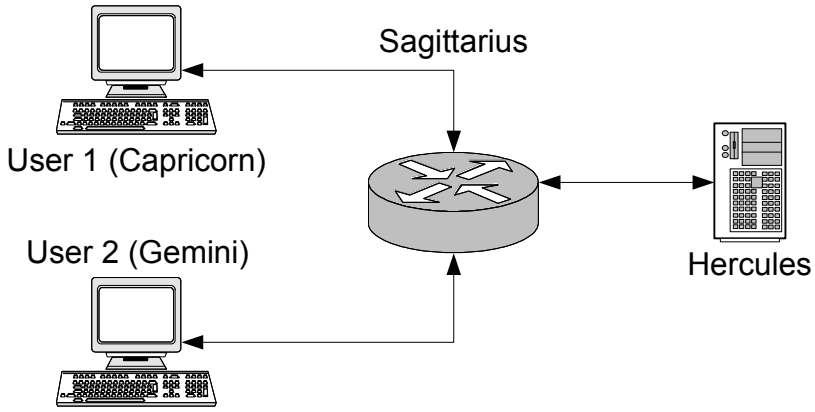

Figure 5.18: Implementation Setup

5.9 Implementation

We also tested the RL based resource reservation and provisioning scheme on a testbed in order to evaluate how difficult it would be to implement such a system in real-world situations. It is a proof-of-concept implementation rather than a full blown one. The testing was done by allowing network bandwidth reservation and CPU provisioning. CPU provisioning was done using the *nice* utility in Linux, which can set the priority of a process. Network reservation was provided using a software known as Rshaper ([71]). We also tried implementing CPU reservation using DSRT ([66]) and RTAI ([67]). However, DSRT seems not to work with newer versions of the Linux kernel, and is not actively being developed. With RTAI, we tried implementing a master scheduler that can assign slices of CPU time to various processes, depending on how much CPU they have reserved. However, we could not get process preemption to work, which meant that the slave processes had to manually wake up the master once their allotted cycles are over. This was deemed impractical since grid jobs would need to implement RTAI specific code in order to run.

5.10 Hardware Details

Figure 5.18 shows the topology that was used to test the RL based agent on a physical testbed. The tested consists of two users, each of them being attached to a machine each. User1 always sends Class 1 job, while User2 sends Class 2 jobs to the Grid Resource (GR), Hercules. All the links shown in the figure are Ethernet connections and they have a maximum bandwidth of 100 Mbps.

The machines, Gemini and Capricorn, run Red Hat Linux ([68]) (kernel version 2.4.20) on a Pentium IV processor, and both have a single network interface. They submit jobs to the grid resource by using the Java CoG toolkit ([69],[70]).

The router, Sagittarius, is a machine similar to Gemini and Capricorn, the only difference being that it has three network interfaces. Sagittarius runs a bandwidth control software known as Rshaper ([71]). Rshaper can be used to control the incoming bandwidths of different hosts or subnets. In this configuration, rshaper limits the total incoming bandwidth of Gemini and Capricorn to 125 KBytes/sec (1 Mbps). We also developed a daemon program that runs on Sagittarius and is capable of accepting reservation rate data from other network entities. When this daemon program receives a reservation request, it modifies the reservation levels for different IPs on the fly with the help of rshaper.

Hercules, the machine acting as the grid resource is also a Pentium IV machine running Debian Linux Sarge ([72]). It runs Globus version 2.4 ([15]) in order to provide grid services. It also runs the RL network agent responsible for managing network bandwidth at Sagittarius. It communicates with the daemon program at Sagittarius using a client-server protocol over TCP/IP. In addition to that, Hercules also runs a CPU Scheduler, which manages provisioning levels for the incoming jobs. CPU provisioning is done using nice levels in Linux. Nice levels in Linux are in the range [-20,20] with -20 being the highest priority a process can obtain. Processes that require more CPU cycles are assigned lower nice levels, while background jobs and other miscellaneous jobs are assigned high nice levels.

5.11 Configuration of the Experiment

Table 5.12 lists the parameters of the jobs that were executed on the testbed. The deadline for a Class 1 job was 15 seconds, and the deadline for a Class 2 job was 50 seconds. Therefore, Class 1 jobs require more resources than Class 2 jobsin order to meet their deadlines. On the completion of a job, it returns to the user who created the job, and the broker running at Hercules calculates the reward or penalty depending on whether the job is completed before of after deadline.

Table 5.12: Characteristics of Jobs in Implementation Setup

Class	Processing Time (seconds)	Network Size (bytes)	Mean Generation Delay (secs)	Number of jobs	Deadline (s)
1	1.5	232972	2.5	1000	15
2	2.5	232972	2.5	1000	50

5.11.1 Network Agent

One of the RL agents running at Hercules is for the purpose of network bandwidth reservation. For this RL agent, the state space is an array of the delays experienced by the two jobs. The state of the agent had two dimensions, one for the response time of the last completed job of each class. The state space varied from from [0,0] to [20,20]. Response times that were greater than 20 seconds were truncated to 20 s. For the action space, the network bandwidth could be reserved in the ranges of 50%, 55%, or 60% in favour of Class 1. We have deliberately chosen to have equal or more bandwidth for Class 1 compared to Class 2 because we know Class 1 demands lower response times, and reducing the number of possible actions speeds up the convergence of the learning algorithm. The reward structure is defined as below -

If deadline is met ($d_i < D_i$)

$$R = \frac{5}{(d_i - \rho_i) + 0.001} \tag{5.2}$$

else

$$R = -10 \tag{5.3}$$

where ρ_i is the completion time of job i. The reward is given in such as above in order to encourage jobs to finish close to their deadlines. If a job finishes too fast, the reward it receives is lower than what it would have received if it had finished closer to the deadline. This ensures that RL agents to not over-provision resources for one class of jobs.

5.11.2 CPU Agent

Another agent running on Hercules is responsible for assigning nice levels to incoming jobs. In our experiments, a running job is continues to run at the same service it is assigned when it enqueued. The reason for this was that changing the nice level would make it difficult to determine which action caused the job to finish within or after the deadline.

Similar to the case for the network agent, the state space is two dimensional, one dimension for the response time of the last completed job of each class. The state space ranged from [0,0] to [20,20]. The action consists of setting the nice levels for both classes of jobs at the next decision epoch. One of the following nice levels could be chosen by the agent - {(-20,-10), (-20,0), (-20,10), (-20,20), (-10,0), (-10,10), (-10,20), (0,10), (0,20), (10,20)}. The first element of the given tuples is the nice level for class 1, and the second element is the nice level for class 2 jobs. A lower nice level leads to the Linux CPU scheduler giving higher preference to that job. Similar to the case for the network agent, we always assign a higher preference to Class 1 jobs, since we know that they demand lower response time

and hence consume more resources.

The reward structure for the CPU Agent is as follows: If deadline is met ($d_i < D_i$)

$$R = \frac{5}{(d_i - \rho_i) + 0.001} \tag{5.4}$$

else

$$R = -10 \tag{5.5}$$

The reward assignment algorithm is similar to the one used for the network agent (Section 5.11.1), and for the same reasons.

5.11.3 Resource Allocation Policies

We ran the experiments with three different resource allocation policies enabled on the agents.

- Default - In this mode, the agent does not perform any resource allocation. The router runs in its default FIFO mode, and the jobs are run at the GR with the default nice level. Neither the router nor the GR give any preference to Class 1 or Class 2 jobs.

- QL - In this case, the agent performs resource allocation according to the optimum policy as determined by the Watkins Q(λ) algorithm.

- SMART - Similar to the previous case, the network agent determines the bandwidth to be allocated to Classes 1 and 2, and the nice levels for the jobs are set according to the optimum policy learnt by the SMART algorithm.

When using QL and SMART, we used function approximation to reduce the amount of memory required to store the value function data. The function approximation used was the CMAC algorithm (Section 2.1.8).

The default scheduling policy is used as a comparison to QL and SMART because it is the out-of-the-box service levels provided to Globus users. Globus

does not yet have a global standard to provide QoS to grid jobs. Providing QoS requires negotiations between users and service providers; it is not yet an autonomous process.

5.12 Results

Table 5.13: Number of Successful Jobs

Class	Default	QL	SMART
1	198	684	649
2	16	412	474

Table 5.14: Average Response Time for Successful Jobs

Class	Default (s)	QL (s)	SMART
1	13.184	10.293	10.061
2	40.922	28.217	26.494

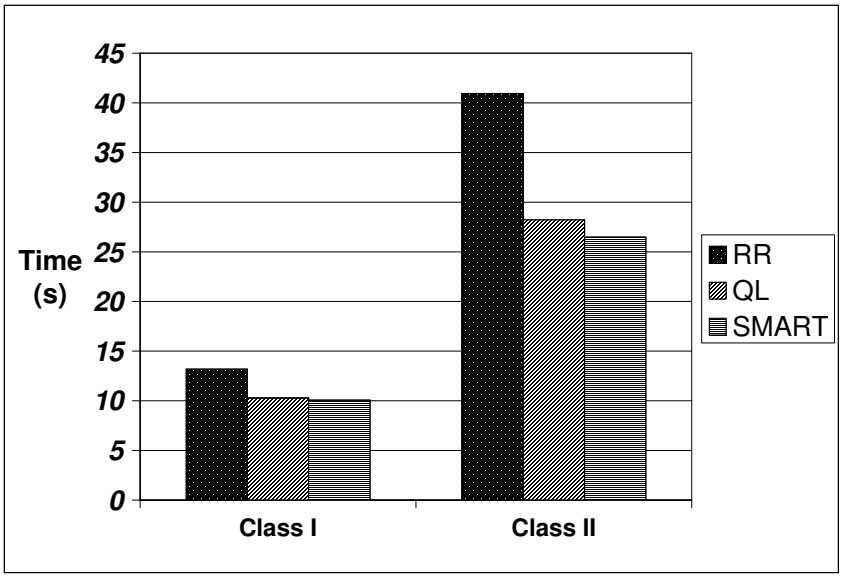

Figure 5.19: Average Response Time of Successful Jobs

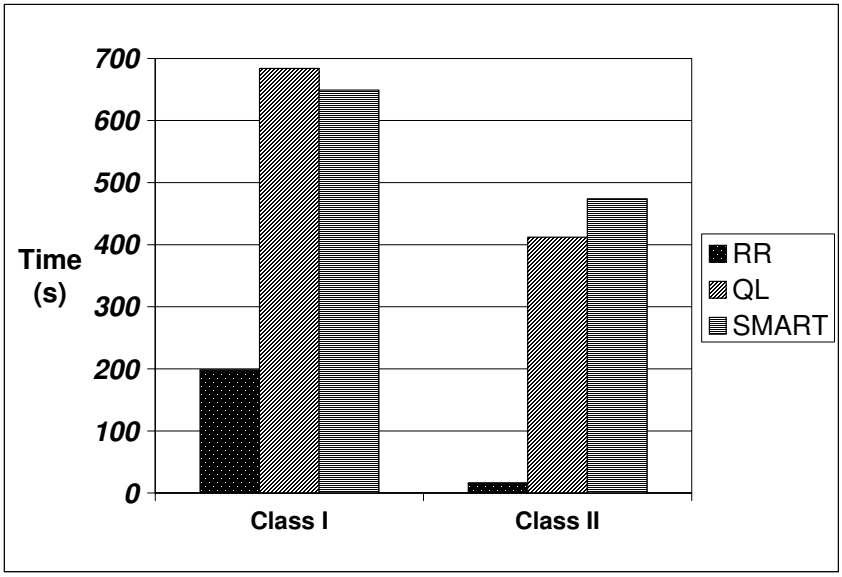

Figure 5.20: Number of Jobs that Finished within their Deadline

Table 5.13 shows the results of running the experiment on the testbed. It shows that the RL based learning algorithm does much better than the default scheduling policy. By allocating CPU cycles and network bandwidth carefully, the RL policies are able to lower response times for both Class 1 and 2. The reason for this was that jobs which failed to meet their deadline were ones that were allocated very few resources, thus freeing up the resources for other jobs. The average response time for jobs that failed to meet their deadline is given in Table 5.15. Some of the jobs also fail because during exploration, the agent can assign some undesirable resource allocations to jobs.

Table 5.15: Average Response Time for Successful Jobs

Class	Default (s)	QL (s)	SMART
1	17.586	20.109	25.186
2	78.384	96.318	105.308

If the QoS mechanisms were work conserving, then one class of jobs could

only perform better at the expense of other classes. This means that in a work conserving system like provisioning, if Class 1 had a worse response time than QL or SMART, then Class 2 would have a better response time. However, we have used reservation for providing QoS at the router, which is a non-work conserving QoS mechanism. In reservation, both classes would only receive their fixed share of bandwidth, regardless of whether the entire bandwidth of the router was free. This is the reason for both Class 1 and 2 jobs having worse response times than QL or SMART.

5.13 Issues with Reinforcement Learning

Agents running the Reinforcement Learning algorithm need to share the reward signal at a minimum. Since the reward itself is a discrete quantity that only needs to be sent at intervals, RL algorithms do not impose a significant load on the network. However, at Grid Resource, the RL algorithms need to maintain value functions for the entire state space and action space. The memory requirements can be reduced with the use of function approximators like CMAC.

Another problem is the training time required for RL. During the exploration phase, actions may be taken that do not produce the desired results. Therefore, a short training time is preferable. However, by using short training times, the RL algorithms may not be able to learn sufficient information in order to perform well in circumstances that did not occur during the training period. Long training periods help the RL algorithms explore more of their state space. The learning time can be reduced significantly with the use of function approximators.

5.14 Conclusions from Simulations and Implementation

From the simulation results in this chapter, we can conclude that reinforcement learning methods are able to adapt successfully to a given scenario. Brokers using RL to learn scheduling policies are able to successfully learn a scheduling policy which works better than Round Robin or Exponential Averaging methods. Resource allocation techniques relying on RL methods are able to modify their allocation levels in such a manner as to support the workload provided to the system. These methods work better than Static methods of resource allocation. Reservation of resources can provide better a guarantee of service than provisioning.

This chapter also presented a proof-of-concept implementation of a reinforcement learning method to allocate resources in order to support QoS. With some training, a RL based resource allocation agent manage a network and grid resources, and requires minimal intervention from the system administrator.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The work presented in this thesis was motivated by the need to provide autonomous Quality of Service mechanisms in grid and utility networks. Providing QoS in such networks is becoming increasingly important as organizations move from in-house processing to renting service from grid service providers. QoS parameters may be negotiated between GSPs and organizations using Service Level Agreements. However, it is cumbersome for GSPs to individually configure resources to match all the various SLAs they have drawn up with their customers.

This thesis explores a Reinforcement Learning based solution to make the management of resources in such scenarios autonomous. The advantage of Reinforcement Learning based methods are that they are self-training, and they can adapt themselves to different scenarios. We explored two different RL methods - Watkins' Q(λ) and SMART, and tested their performance against currently used static provisioning methods. We believe that RL based resource allocation strategies can provide the solution to one aspect of the QoS question, i.e. providing autonomous resource allocation strategies that adapt to the environment at grid and network service providers. We tested our RL methods in simulation to verify

their performance. We also provided a sample implementation to test the viability of our solution, and compared the implementation to other resource allocation strategies.

From the simulation and implementation results, we can conclude that reinforcement learning methods are able to adapt successfully to a given scenario. Brokers using RL to learn scheduling policies are able to successfully learn a scheduling policy and provide QoS. Resource allocation techniques relying on RL methods are able to modify their allocation levels in such a manner as to support the workload provided to the system. These methods work better than Static methods of resource allocation. We can also conclude that reservation of resources can provide better a quality of servicve then provisioning methods.

An advantage of using the Q(λ) or SMART algorithm is that both are model free. A model is essentially an agent's view of the environment, which maps state-action pairs to probability distributions over states. When the possible number of state-action pairs increase, the memory requirement of a model-based algorithm will increase proportionately. In grid environments, where a large number of resources should be expected, the memory requirement will be quite high. The model free algorithms we have used have much lesser memory requirements due to the lack of a model, and hence should be scalable to real-world grid networks.

RL based systems require a certain amount of time to learn their policy well. During this time, some of the decisions may lead to worse behaviour than static policies, due to the exploration that the agent is carrying out. In order to cut down on the learning time, we can train the agent in a simulation environment which resembles the environment the agent is expected to operate in. After this, the agent may be used in the real system, and due to its previous learning, should be able to perform much better.

6.2 Contributions

This thesis presents a scheme for resource allocation in Grids and network domains via Reinforcement Learning algorithms. The RL based agents are able to adjust the resource allocation levels without intervention from a supervisor, and it can adapt itself to any scenario without the need for a model of the problem.

Our major contribution in this thesis is the design and implementation of a RL based system to achieve QoS paramaters for users with different requirements. We provide a comprehensive analysis of our solution in simulation, and verify the results with an implementation on a Globus based testbed. From our study, we concluded that the RL methods would be a viable technique to use on grid networks for resource allocation.

We also contributed a network simulation package that works with GridSim. This will help other researches simulate their own various proposals to improve the design of grid networks.

6.3 Recommendations for Future Work

6.3.1 Co-ordination among Agents

In our study, the agents are completely independent of each other, and their actions are taken as if no other agents exist in the system. This may lead to slower convergence due to conflicting actions of the agents. For example, when a job fails, both the routers in between and the grid node which processed the job receive a negative reward. This will cause agents on both of them to increase allocation levels for the class to which the failed job belongs. If the job failed by missing the deadline quite narrowly, it may be possible that increasing only network bandwidth reservation or CPU reservation would have been enough. Therefore, it would be better if there was some kind of co-ordination among the agents managing these

resources.

One method in which agents can co-ordinate is to present the bandwidth and processing allocation problem as a joint problem. However, this approach leads to an increase in both state space and action space, which increases the size of the problem exponentially.

Rather than following a brute-force merging of state and action space technique, an option would be to try implementing Co-ordinated Reinforcement Learning ([73]). This approach is based on approximating the joint value function as a linear combination of value functions of the individual agents. Agents communicate reinforcement learning signals, utility values and policies in order to achieve convergence quickly and correctly. [73] provides an example of implementing Co-ordinated RL to a Q-learning problem, and shows how such communication and co-ordination can be achieved.

6.3.2 Better Network support in GridSim

GridSim supports network elements after our work, but there are a lot more features that need to be incorporated in order for it to grow further as a simulation package. Currently, GridSim only supports having a single link to an entity that is not a router. It would be desirable to have multiple links and routing tables which support multiple routes to a destination. With this implemented, brokers can not only choose which grid node their jobs should run on, they can also choose the route to be taken to the node. Note that this is not currently not possible even on the public Internet, where a user has no control over his packets once they leave his Administrative Domain (AD).

It would also be desirable to support features like finite buffers and TCP like mechanisms to retransmit dropped packets. The addition of these features would make GridSim a comprehensive grid simulation package.

Bibliography

[1] I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *International Journal of High Performance Computing Applications*, vol. 15, no. 3, pp. 200–222, 2001.

[2] D. Werthimer, J. Cobb, M. Lebofsky, D. Anderson, and E. Korpela, "SETI@HOME - Massively Distributed Computing for SETI," *Computer Science Engineering*, vol. 3, no. 1, pp. 78–83, 2001.

[3] "Distributed.net." [Online]. Available: http://www.distributed.net

[4] V. Berstis, *Fundamentals of Grid Computing.* IBM Redbooks, 2002. [Online]. Available: http://www.redbooks.ibm.com/redpapers/pdfs/redp3613.pdf

[5] D. A. Menascé and E. Casalicchio, "Quality of Service Aspects and Metrics in Grid Computing," in *Proceedings of the Computer Measurement Group Conference*, Las Vegas, Nevada, USA, December 7–10 2004.

[6] D. Lin and R. Morris, "Dynamics of Random Early Detection," in *SIGCOMM '97*, Cannes, France, september 1997, pp. 127–137.

[7] "Quality of Service (QoS) Networking." [Online]. Available: http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/qos.htm

[8] J. Heinanen, T. Finland, F. Baker, W. Weiss, and J. Wroclawski, "RFC 2597: Assured Forwarding PHB Group," June 1999. [Online]. Available: http://rfc.net/rfc2597.html

[9] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, "RFC 2475: An Architecture for Differentiated Service," December 1998. [Online]. Available: http://www.ietf.org/rfc/rfc2475.txt

[10] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin, "RFC 2205: Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification," September 1997. [Online]. Available: http://www.faqs.org/rfcs/rfc2205.html

[11] S. Shenker, C. Partridge, and R. Guerin, "RFC 2212: Specification of Guaranteed Quality of Service," September 1997. [Online]. Available: http://www.faqs.org/rfcs/rfc2212.html

[12] "Integrated Services Charter." [Online]. Available: http://www.ietf.org/html.charters/intserv-charter.html

[13] X. Xiao and L. M. Ni, "Internet QoS: A Big Picture," *IEEE Network*, vol. 13, no. 2, pp. 8–18, Mar. 1999.

[14] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy, "A Distributed Resource Management Architecture that Supports Advance Reservations and Co-allocation," in *In International Workshop on Quality of Service: (IWQoS 99)*. IEEE Press, June 1999, pp. 27–36, http://www.globus.org/documentation/incoming/iwqos.pdf.

[15] "Globus toolkit." [Online]. Available: http://www.globus.org/

[16] I. Foster, A. Roy, V. Sander, and L. Winkler, "End-to-End Quality of Service for High-End Applications," Argonne National Laboratory, Tech. Rep., 1999.

[17] R. Buyya, "Economic-Based Distributed Resource Management and Scheduling for Grid Computing," PhD. thesis, Monash University, Melbourne, Australia, 2002.

[18] P. V.-B. Primet and F. Chanussot, "End-to-End Network Quality of Service in Grid Environments: The QoSINUS approach," in *Workshop on Networks for Grid Applications, BROADNETS 2004*, San José, California, USA, October 25–29 2004.

[19] R. J. Al-Ali, K. Amin, G. von Laszewski, O. F. Rana, and D. W. Walker, "An OGSA-Based Quality of Service Framework." in *GCC (2)*, ser. Lecture Notes in Computer Science, M. Li, X.-H. Sun, Q. Deng, and J. Ni, Eds., vol. 3033. Springer, 2003, pp. 529–540.

[20] K. Nahrstedt, H. H. Chu, and S. Narayan, "QoS-aware Resource Management for Distributed Multimedia Applications," *Journal of High Speed Networking*, vol. 7, no. 3-4, pp. 229–257, 1999.

[21] A. Galstyan, K. Czajkowski, and K. Lerman, "Resource Allocation in the Grid Using Reinforcement Learning," in *International Conference on Autonomous Agents and Multiagent Systems*, Columbia University, New York City, USA, 2004.

[22] D. Vengerov, "A reinforcement learning framework for utility-based scheduling in resource-constrained systems," Sun Microsystems Labs Institute, Menlo Park, Tech. Rep. TR-2005-141, 2005.

[23] R. E. Bellman, *Dynamic Programming*. Dover Publications, 2003.

[24] R. S. Sutton, "Learning to Predict by the Methods of Temporal Differences," *Mach. Learn.*, vol. 3, no. 1, pp. 9–44, 1988.

[25] C. Watkins, "Learning from Delayed Rewards," PhD. thesis, King's College, Cambridge, United Kingdom, 1989.

[26] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA: MIT Press, 1998, a Bradford Book. [Online]. Available: http://www-anw.cs.umass.edu/~rich/book/the-book.html

[27] L. P. Kaelbling, M. L. Littman, and A. P. Moore, "Reinforcement Learning: A Survey," *Journal of Artificial Intelligence Research*, vol. 4, pp. 237–285, 1996.

[28] D. P. Bertsekas and J. N. Tsitsiklis, *Neuro-Dynamic Programming*. Athena Scientific, 1996.

[29] G. Tesauro, "Practical Issues in Temporal Difference Learning," in *Advances in Neural Information Processing Systems*, J. E. Moody, S. J. Hanson, and R. P. Lippmann, Eds., vol. 4. Morgan Kaufmann Publishers, Inc., 1992, pp. 259–266.

[30] C. K. Tham, "Modular On-Line Function Approximation for Scaling up Reinforcement Learning," PhD. thesis, Cambridge University, 1994.

[31] C. J. C. H. Watkins and P. Dayan, "Technical Note: Q-Learning," *Machine Learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[32] T. K. Das, A. Gosavi, S. Mahadevan, and N. Marchalleck, "Solving Semi-Markov Decision Problems Using Average Reward Reinforcement Learning," *Management Science*, vol. 45, no. 4, pp. 560–574, 1999.

[33] S. P. Singh and R. S. Sutton, "Reinforcement Learning with Replacing Eligibility Traces," *Machine Learning*, vol. 22, no. 1–3, pp. 123–158, 1996.

[34] M. L. Puterman, *Markov Decision Processes : Discrete Stochastic Dynamic Programming*. New York, USA: Wiley Interscience, 1994.

[35] C. Darken, J. Chang, and J. Moody, "Learning Rate Schedules for Faster Stochastic Gradient Search," in *Proc. Neural Networks for Signal Processing 2.* IEEE Press, 1992.

[36] A. Sulistio, G. Poduval, R. Buyya, and C.-K. Tham, "Constructing A Grid Simulation with Differentiated Network Service Using GridSim," in *Proceedings of the The 2005 International MultiConference in Computer Science & Computer Engineering (ICOMP '05)*, Las Vegas, Nevada, USA, 2005.

[37] I. Foster and C. Kesselman, Eds., *The Grid: Blueprint for a Future Computing Infrastructure.* Morgan Kaufmann Publishers, 1999.

[38] R. Buyya and 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*, vol. 14, no. 13–15, pp. 1175–1220, Nov./Dec. 2002.

[39] "GridSim Web Resource." [Online]. Available: http://www.gridbus.org/gridsim/

[40] R. Buyya, D. Abramson, and J. Giddy, "Nimrod-G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid," in *Proc. of the 4th International Conference and Exhibition on High Performance Computing in Asia-Pacific Region (HPC Asia'00)*, Beijing, China, May 14–17 2000.

[41] C. Simatos, "Making SimJava Count," MSc. Project report, The University of Edinburgh, September 12 2002.

[42] F. Howell and R. McNab, "SimJava: A Discrete Event Simulation Package for Java with Applications in Computer systems Modeling," in *Proc. of the 1st International Conference on Web-based Modelling and Simulation*, San Diego, USA, January 11–14 1998.

[43] M. Priestley, *Practical Object-Oriented Design with UML.* McGraw-Hill, 2000.

[44] G. Malkin, "RFC 2453: RIP version 2," November 1998. [Online]. Available: http://www.apps.ietf.org/rfc/rfc2453.html

[45] J. Moy, "RFC 2328: OSPF version 2," April 1998. [Online]. Available: http://www.ietf.org/rfc/rfc2328.txt

[46] J. Postel, "Internet Control Message Protocol: DARPA Internet Program Protocol Specification," September 1981. [Online]. Available: http://www.ietf.org/rfc/rfc0792.txt

[47] A. J. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," in *Proc. of the ACM Symposium on Communications Architectures and Protocols (SIGCOMM'89)*, Austin, USA, September 19–22 1989, pp. 1–12.

[48] S. J. Golestani, "A Self-Clocked Fair Queueing Scheme for Broadband Applications," in *Proc. of IEEE INFOCOM'94*, Toronto, Canada, June 12–16 1994, pp. 636–646.

[49] "The Network Simulator – ns-2." [Online]. Available: http://www.isi.edu/nsnam/ns/

[50] J. Liu and D. M. Nicol, *DaSSF 3.1 User's Manual*, Dartmouth College, April 2001.

[51] A. Varga, "The OMNeT++ Discrete Event Simulation System," in *Proc. of the European Simulation Multiconference (ESM'01)*, Prague, Czech Republic, June 6–9 2001.

[52] "J–Sim." [Online]. Available: http://www.j-sim.org/

[53] K. Aida, A. Takefusa, H. Nakada, S. Matsuoka, S. Sekiguchi, and U. Nagashima, "Performance Evaluation Model for Scheduling in a Global Computing System," *The International Journal of High Performance Computing Applications*, vol. 14, no. 3, pp. 268–279, 2000.

[54] H. J. Song, X. Liu, D. Jakobsen, R. Bhagwan, X. Zhang, K. Taura, and A. Chien, "The MicroGrid: a Scientific Tool for Modeling Computational Grids," in *Proc. of IEEE Supercomputing 2000*, Dallas, USA, November 4–10 2000.

[55] X. Liu and A. Chien, "Realistic large-scale online network simulation," in *Proc. of IEEE Supercomputing 2004*, Pittsburgh, USA, November 6–12 2004.

[56] H. Casanova, "Simgrid: A Toolkit for the Simulation of Application Scheduling," in *Proc. of the First IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid'01)*, Brisbane, Australia, May 15–18 2001.

[57] A. Legrand, L. Marchal, and H. Casanova, "Scheduling Distributed Applications: The SimGrid Simulation Framework," in *Proc. of the Third IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid'03)*, Tokyo, Japan, May 12–15 2003.

[58] W. H. Bell, D. G. Cameron, L. Capozza, A. P. Millar, K. Stockinger, and F. Zini, "OptorSim – A Grid Simulator for Studying Dynamic Data Replication Strategies," *The International Journal of High Performance Computing Applications*, vol. 7, no. 4, pp. 403–416, 2003.

[59] "Bricks: A Performance Evaluation System for Grid Computing Scheduling Algorithms." [Online]. Available: http://www.is.ocha.ac.jp/~takefusa/bricks/index.shtml

[60] I. Foster and C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit," *The International Journal of Supercomputer Applications and High Performance Computing*, vol. 11, no. 2, pp. 115–128, 1997.

[61] R. Wolski, N. Spring, and J. Hayes, "The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing," *The Journal of Future Generation Computing Systems*, vol. 15, no. 5–6, pp. 757–768, 1999.

[62] A. Demers, S. Keshav, and S. Shenker, "Analysis and Simulation of a Fair Queueing Algorithm," in *SIGCOMM '89: Symposium proceedings on Communications architectures & protocols*. New York, NY, USA: ACM Press, 1989, pp. 1–12.

[63] A. K. Parekh and R. G. Gallager, "A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-Node Case," *IEEE/ACM Trans. Netw.*, vol. 1, no. 3, pp. 344–357, 1993.

[64] H. Zhang and D. Ferrari, "Rate-Controlled Service Disciplines," *High-Speed Networks*, vol. 3, no. 4, 1994.

[65] H. Zhang, "Providing end-to-end performance guarantees using non-work-conserving disciplines," *Computer Communications*, vol. 18, no. 10, Oct 1995.

[66] R. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling,*. John Wiley and Sons, Inc., May 1991, winner of "1991 Best Advanced How-To Book, Systems" award from the Computer Press Association.

[67] A. Rubini, "Rshaper." [Online]. Available: http://www.linux.it/~rubini/software/#rshaper

[68] H.-H. Chu, "CPU Service Classes: A Soft Real Time Framework for Multimedia Applications," University of Illinois, Urbana, Illinois, Tech. Rep. 2106, 1999.

[69] L. Dozio and P. Mantegazza, "Real Time Distributed Control Systems Using RTAI." in *6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2003).* IEEE Computer Society, 2003, pp. 11–18.

[70] "RedHat Linux." [Online]. Available: http://www.redhat.com/

[71] "CoG Kit – Java." [Online]. Available: http://www-unix.globus.org/cog/java/

[72] "Grid Job submission using the Java CoG Kit." [Online]. Available: http://www-106.ibm.com/developerworks/library/ws-gridcog.html?ca=dgr-gridw09GridCoGkit

[73] "Debian Sarge." [Online]. Available: http://www.debian.org/releases/stable/

[74] C. Guestrin, M. G. Lagoudakis, and R. Parr, "Coordinated Reinforcement Learning," in *ICML '02: Proceedings of the Nineteenth International Conference on Machine Learning.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002, pp. 227–234.