QOS mechanisms for software services on grids

B.Tech Project

Submitted in partial fulfillment of the requirements
for the degree of

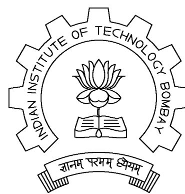
Bachelor of Technology

by

Devesh Sukhwal
Roll No: 00005002

under the guidance of

Prof. Varsha Apte



Department of Computer Science and Engineering
Indian Institute of Technology
Bombay
May 11, 2004

Abstract

The project aims to develop mechanisms for providing QOS on computational grids, wherein large computing systems are given on lease for many applications, thereby deriving the benefits of large scale. The main problems facing scientists in this regard are of effective scheduling, resource allocation and admission control. Although experimental grids are being setup at various labs, we need better admission control mechanisms. We focus on developing strategies for better admission control in this project, and setting up prototypes for testing the mechanism developed.

1 Introduction

With computing industry getting more and more professional by the day, the old concepts of each application host maintaining his own computational resources are no longer valid. Instead the trend is shifting towards taking computing power on lease, and concentrating on the mainline business.

Admission control can be of two types. One is admission of a new request by an existing client with whom we have signed a contract(here we will refer to it as traffic control). The other is the admission of a contract with a new client(we will call it server admission control).

Besides the obvious economies of large scale, Computational Grids have numerous other advantages. First, many large scale applications e.g. in molecular biology need enormous resources which cannot be provided by a single machine so they have to use grids. Second, many organizations need computing power only once in a while, and not always; so delegation of the process of setting up and maintaining computing resources seems to be a good idea. Service providers do not have to keep resources to fulfill their worst case demands, because it is not likely that all their customers need their maximum requirements at the same time.

For this to be successful, computational grid providers will have to give some quality of service guarantees. For example, the server may guarantee that if the client makes less than 10 requests per minute, all the requests will be completed less than 10 seconds. As the systems become more and more complex, the QOS specification may also get complicated. For instance, the number of requests per unit time may not correspond to the cost to the server, because there may be a few, but more demanding requests. QOS contract specification based on utilization of the resources is also not advisable because the resources are many(e.g. processor, network bandwidth, disk accesses etc.) and the client may not be able to quantify the hardware utilization requirements of his requests. Somewhere we have to strike a balance between the two approaches.

Doing this becomes all the more difficult because processes often need resources in a lump, and a partial procurement of the resources is useless. For example, a process may request some disk accesses, and some processing power. Either of the two are useless without the other. Thus, bundling the resources together, and when the process has finished, re-bundling for some other process adds to the challenges.

In the project we analyzed some computational grid systems which have been designed, especially the ones which provide a shared hosting platform i.e. where the system is shared by many applications of diverse nature and job requests are sent over a network. We have also developed a testbed for trying, testing and evaluating new strategies; and evaluated some sample admission control strategies(of the above mentioned two types).

The rest of this report is organized as follows. In section 2 we give an introduction to modelling the resource allocation problem in grids as a free market. In section 3, we discuss a strategy which extends the market model, and also suggests a new way of representing QOS mathematically. In section 4 we propose a solution similar to the latter, adding to its admission control technique. In section 5, we have described the platform on which the techniques will be tested and evaluated. Testing methods and results are described in 6, and we summarize in section 7.

2 Resource Allocation based on a economic models

Since the dynamics of a Grid Computing system are difficult to model, the system is modelled as a market economy ([RWB01], [KS02]) and hence the past research in the field of economics can be put to good use here. The various resources in the grid system(e.g. CPU, bandwidth etc.) are modelled as hypothetical resource producers, who "sell" their resources to customers i.e. hypothetical resource consumers. The incoming job requests come with a "budget" allocated(a measure of the client perceived value of the job). They may have to wait longer, or are starved if prices of resources are too high for their budget. Two types of strategies are described in [RWB01] commodity markets and auctions. It is assumed that relative worth of a resource is determined by the demand and supply for it. As in a market, success of a particular strategy is measured in terms of four factors :-

1. **Price stability** :- Schedulers base their decisions on the state of economy as suggested by price of different commodities. If this varies wildly, their decisions will not be effective leading to poor performance.

2. **Equilibrium** :- The state of an economy when the demand for the commodities is same as the supply of it. The price of a commodity, as in a real market, reflects its relative worth only if the economy is in a state of equilibrium. Therefore if the market cannot be brought in equilibrium, decisions will be poor.

3. **Application Efficiency** :- The measure of how effective the grid is as a computing platform.

4. **Resource Efficiency** :- The measure of how effectively the grid manages its resources.

2.1 Commodity Markets

In every period, a CPU producer agrees to sell a fixed number of "slots" to the grid. A faster CPU will be able to sell more slots per unit time than a slower one. Similarly all resource producers declare their slots. To determine supply at a given point, each producer keeps track of a quantity mean_price, which is the average price per unit time, at which they have sold their resources so far.

$$mean_price = (revenue/elapsed_time)/slots$$

where, *revenue* is the total amount of money this producer has made in *elapsed_time*. Producers sell only if the current price exceeds this mean_price. If a producer cannot sell in some time period, its mean price comes down and it is more likely to sell in the next period hence *mean_price* reflects the demand for the resource of a producer.

Consumers express their need to the market in the form of jobs, e.g. when they need disk for 100 minutes, and CPU for 200 minutes they declare only their need for one CPU slot and one disk slot without revealing the duration. They are given an initial budget and a periodic allowance but are not allowed to hold the money from one period to the next. Also, if consumers cannot afford one of the resources they need, they do not express their need for the resource of other type also. To determine supply, the consumers calculate *avg_rate*, and *capable_rate* as follows.

$$avg_rate = \frac{\sum total_work_i * price_i}{now}$$

$$capable_rate = \frac{remaining_budget}{refresh - now}$$

where, $total_work_i$ is total work performed using commodity i,
$price_i$ is the current price for commodity i,
$remaining_budget$ is total budget remaining to spend before budget refresh,
$refresh$ is the next refresh time, and
now is the present time
Demand is expressed only if $capable_rate$ exceeds avg_rate.

If prices of commodities are expressed as a price vector $\mathbf{p} = (p_1\, p_2\, ...\, p_n)$, where p_i stands for price of the ith commodity. Excess demand z_j for the jth commodity is defined as the demand minus supply. It may be positive, or negative. Since the markets for all the commodities are related, each z_j is a function of all the prices. An equilibrium point $\mathbf{p}^*$ such that $\mathbf{z}(\mathbf{p}^*) = 0$ must exist, by Smale's theorem([RWB00]. Also, for any value of $\mathbf{p}$ we can form the n×n matrix of partial derivatives

$$D_z(\mathbf{p}) = (\frac{\partial z_i}{\partial p_j})$$

The above equation can be solved(approximately) for $\mathbf{z}(\mathbf{p}) = 0$ ([RWB00]).

2.2 Auctions

Resource producers auction themselves using a centralized auctioneer, and sealed-bid auctions. When there is only one type of resources to be desired, auction is a convenient way to allocate resources. But when there are more than one, the consumers have to place simultaneous bids in more auctions, and may succeed in only a few of these. So, it must spend its money to hold these resources while waiting for others.

At each time step, the producers declare their available resource slots, along with the minimum selling price(the $mean_price$ as described in 2.1). Consumers decide their bid price based on the demand function(as described in 2.1 above). If there is no second-highest bidder, the price of the commodity is the average of the minimum selling price, and the consumer's bid. If a consumer cannot get all of its resources, it still has to pay for the few which it got.

2.3 Results of Simulation

Commodity markets perform better than auctions in most of the factors mentioned above. Since there are more than one type of resources involved, commodity markets are the natural choice. But auctions are simpler to implement, and widely studied.

Efficiency Metric	Under-demand	Over-demand
Consumer jobs/min	0.07	0.03
CPU Utilization	35.2%	85.5%
Disk Utilization	37.6%	85.1%

Table 1: Auctions : Consumer and Product Efficiencies (from [RWB01])

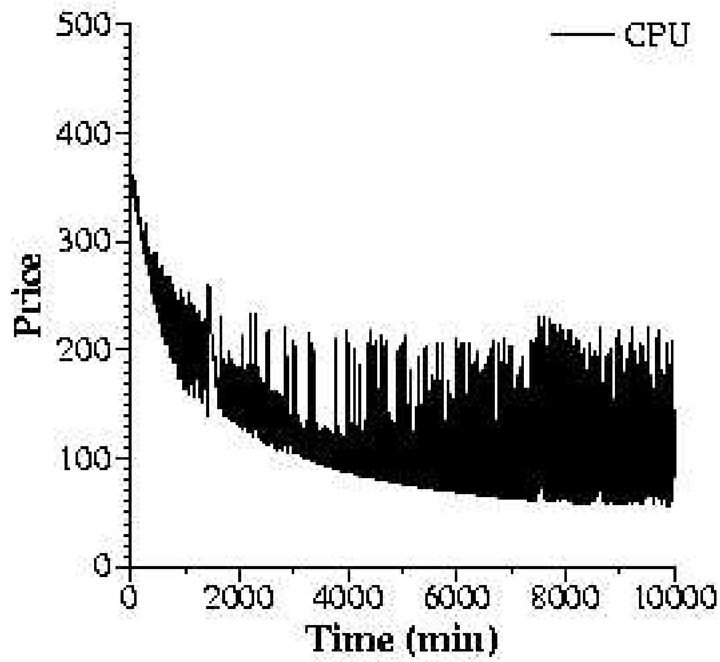


Figure 1: Auction prices for the under demand case, average CPU price only (from [RWB01])

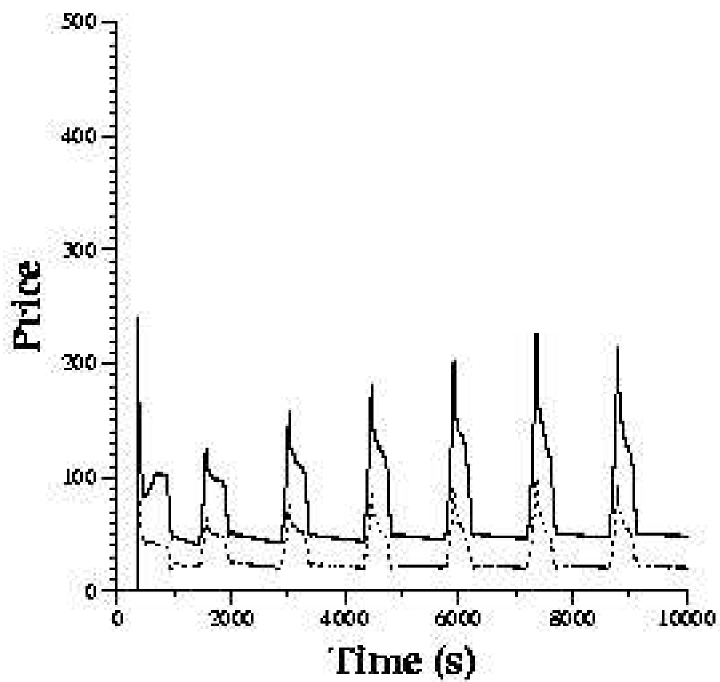


Figure 2: Prices for the under-demand case. Solid line is CPU price, and dotted line is disk price (from [RWB01])

Efficiency Metric	Under-demand	Over-demand
Consumer jobs/min	0.13	0.04
CPU Utilization	60.4%	93.9%
Disk Utilization	54.3%	84.6%

Table 2: Commodity Markets : Consumer and Product Efficiencies (from [RWB01])

3 QOS guarantees, and differentiated service

In the simple economic models discussed in section 2, it was not possible to give a guarantee of any kind, and there was no quantification of QOS. We address those issues here, and discuss ways of giving a money back guarantee.

Since we do not want one misbehaving customer to eat up the resources which could be used to fulfill another's guarantees, we need to segregate the requests, and give per-service-class guarantees. This will prevent starvation of low-priority customers during sustained overload periods. Since there can be moments when the load becomes higher than which can be handled by the server, the server must adapt gracefully to such transient overload. In such circumstances it may have to degrade the QOS given to clients, so the contract needs to be flexible to allow for transient overload at the server. At the same time, different clients may demand different types of service, and some customers may be more important for the server than others. Hence, similar to the economic models, after all it is the server's revenue which has to be maximized ([AS98], [TAB03]).

[AS98] suggest a strategy for resource allocation in a grid computing system and claim to meet all of the above requirements. They introduce the concept of a flexible QOS contract. Here, for each client there are multiple QOS levels each associated with certain service characteristics, and the amount of revenue the server receives if this level of service is provided, which is commensurate with the client perceived utility of receiving service at that level. Under overload, the system can degrade the service given to a client predictably as per the contract(also lowering its own revenue), but if the service is degraded below the lowest QOS level, the server has to pay a QOS violation penalty.

3.1 Representation of QOS

A QOS level is represented as $C_i = (M_i, P_i, B_i)$. At the sender side it means that the resources are enough to send M_i number of bytes every P_i units of time. B_i is the bucket size i.e. the maximum number of bytes to be buffered due to the sender's burstiness (assuming B_i is specified in multiples of M_i). In any interval t, the maximum number of bytes the sender can generate is $N_{max}(t) = (B_i + \lfloor t/P_i \rfloor) * M_i$. $\lfloor t/Pi \rfloor$ bytes will have been transmitted by the end of time t, maximum number of bytes waiting at any time is $B_i * M_i$. A byte generated at time t is said to be **conformant** if total number of bytes generated till time t is less than $N_{max}(t)$. So a byte generated at time t, has to be sent before time $D_i = (t + B_i P_i)$ to give a service of level i. The guarantees are given only to the conformant bytes.

At the receiver side, service of level C_i means that the reserve holds enough resources to receive M_i bytes every P_i time units. B_i is the size of the delivery buffer.

3.2 Optimization of server revenue

Each client, or service class is handled by a separately schedulable entity, an adaptive negotiation agent(ANA). Created at the time of connection setup, it expresses the client's contract terms. To maximize the reward, the design utilizes a combination of 2 modules.

QOS level			Reward
M_1	P_1	B_1	R_1
M_2	P_2	B_2	R_2
	...		
M_m	P_m	B_m	R_m
QOS violation penalty = V			

Table 3: Every client has a similar contract

1. A **Load Control module**, implementing a QOS-optimization algorithm that maximizes total revenue, using an estimate of current load. Activated when a new ANA is to be created or destroyed. Here, it performs an admission test on the new ANA and recomputes the active QOS level for all current ANAs to adapt to the current load. The time complexity of this QOS optimization algorithm is high, and hence cannot be used very frequently.

2. A **Monitoring module** which measures and updates estimated load parameters. The server should adapt to transient load and resource-capacity changes gracefully. Since QOS-optimization algorithm cannot be called repeatedly for it can consume resources, a fast-feedback loop implemented by the monitoring module is provided, that detects transient conditions and makes small incremental adjustments to the QOS levels using a simple heuristic(section 3.4).

3.3 Load Control Module

Load control module is the subsystem which is responsible for the admission control, and the near-optimal selection of QOS levels for clients, such that aggregate reward is maximized. Suppose the subsystem has a fraction U_{server} of the CPU utilization, and a bandwidth BW. Consider n ANAs, the ith ANA having m_i QOS levels. The kth level in the ith ANA is $L_i[k] = (M_i[k], P_i[k], B_i[k])$ for which the reward is $R_i[k]$, and the QOS violation penalty for this ANA is V_i. Here, we introduce a new(artificial) k^{th} level(say null level) such that it has no resource requirements, and has a reward equal to the negative of the QOS violation penalty for this ANA.

$$M_i[k] = 0, P_i[k] = \infty, B_i[k] = 0, \; and \, R_i[k] = -V_i$$

Cost function $f_c(x)$ of the system is defined as the time required to process and transmit x bytes. So the periodic execution cost $E_i[k] = f_c(M_i[k])$. A QOS level $L_i[k]$ requests a CPU utilization $U_i = E_i[k]/P_i[k]$, since $E_i[k]$ is the amount of execution(in cpu-time units) which must complete within $P_i[k]$ time units. The condition for the QOS level to be feasible is

$$U_i[k] < U_{server}, \;\; and \;\; BW_i[k] = (M_i[k]/P_i[k]) < BW$$

This optimization problem is NPC, because it is a derivative of the knapsack problem. To convert it into a polynomial time algorithm, we assume that U_i can take only discrete values, i.e. only integral multiples of some δ. S(i,U) is defined to be the function to solve the subproblem of optimally assigning QOS levels to the i ANAs, without exceeding some utilization U of the server, and returns the optimal reward. Also, let BW(i,U) be the function returning the bandwidth consumed in the QOS levels selected above. We denote the whole set of subproblems S(i,δ), S(i,2δ), ... S(i,U_{server}) as S(i,*). It can be solved using dynamic programming, and the recursive relation

$$\text{S(i,U)} = max_{1 \leq k \leq m_i}\{R_i[k] + S(i-1, U - U_i[k]) \mid BW_i[k] + BW(i-1, U_i[k]) \leq BW\}$$

If k^* is the value of k which yields the computed maximum, the bandwidth will be given by

$$BW(i, U) = BW_i[k^*] + BW(i-1, U - U_i[k^*])$$

The complete algorithm is given below.

1. for i = 1 to n
2. for U = 0 to U_{server} in steps of δ
3. compute S(i,U)
4. compute BW(i,U)
5. S(n, U_{server})

3.4 Monitoring Module

Since the above algorithm is expensive, this monitoring module profiles the system and detects overload and under-load, and then makes greedy incremental adjustments to the QOS levels of customers periodically.

Overload can be detected on failure of d scheduled ANAs to meet their deadlines, where d can be adjusted according to the criticality of the connection(lower for urgent, mission critical services). On detection of overload, choose the ANA C_i whose degradation by one QOS level would result in the minimum decrease in reward, and decrease its level by one.

Under-load can be detected on falling of resource utilizations below a certain threshold. On detection of under-load, choose the ANA C_i whose promotion by one QOS level would result in the maximum increase in reward, and increase its level by one.

3.5 Experiments, and Results

In the experiment, system response to transient load disturbances was analyzed. The number of connections were fixed, multiple QoS levels defined for each connection, then load on the host was varied letting the QoS-optimization algorithm run every 5 seconds to recompute the QoS levels based on the most recent estimate of effect. Two long compilation tasks were started concurrently with packet transmission to overload the CPU. Figure shows the results of a representative run. The top part of the figure shows 5 connections, labeled C1 ... C5, where C1 is the least important connection, and C5 is the most important. The QoS contract for each connection had 3 QoS levels of bandwidth 1Mb (M_i = 100kb; P_i = 100ms), 0.33Mb (M_i = 67kb; P_i = 200ms) and 0.11Mb (M_i = 33kb; P_i = 300ms) respectively. Rewards were assigned proportionally to the bandwidth and weighted by connection importance. Thus, the reward for QoS level k of connection C_i was $R_i[k] = iM_i[k]$.

The figure depicts the QoS level selected for each connection at every invocation of the load control module. The bottom part of the figure shows the change in the measured cost-per-packet, γ_{effect}, as well as the number of missed deadlines between successive invocations of the load control module. (A deadline miss means that M_i bits weren't transmitted within P_i time units.) As can be seen from the figure, less important connections were degraded during overload intervals to keep aggregate system throughput below saturation. The aggregate reward was thus significantly higher than in the case of indiscriminate degradation (where all of the connections would have been degraded below the nominal QoS level).

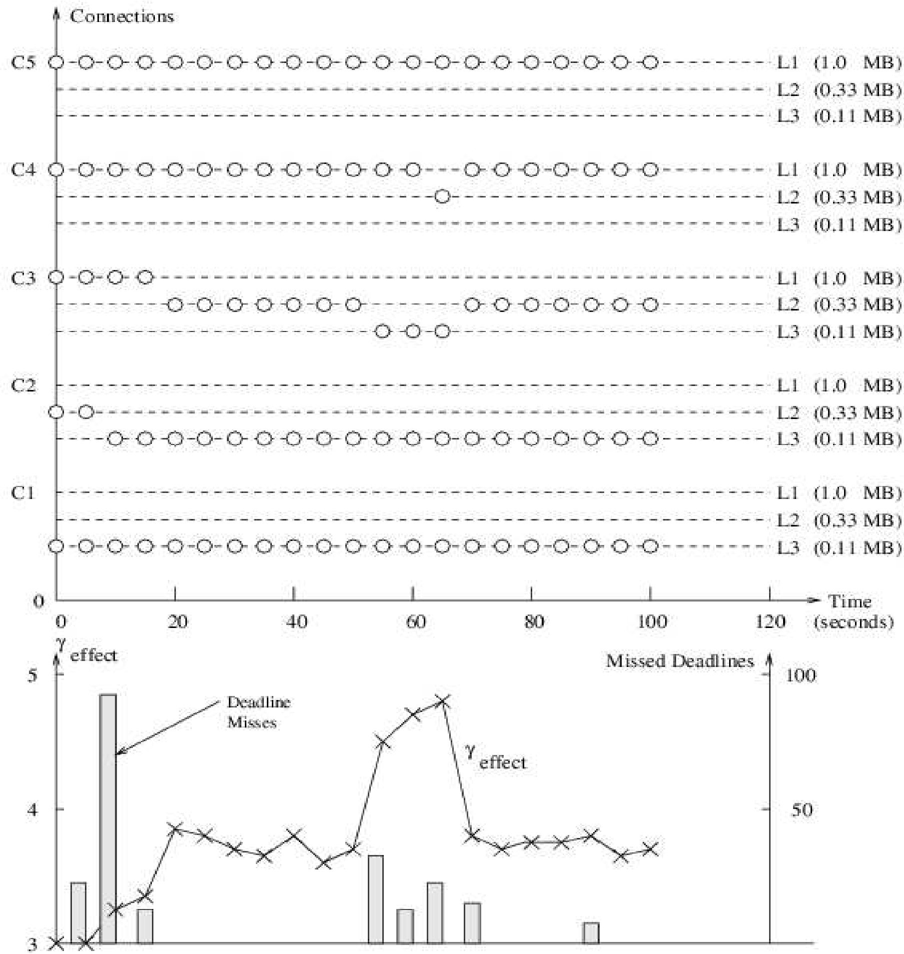


Figure 3: QOS level adaptation on transient load disturbances (from [AS98])

4 The solution attempted in this project

To define any resource allocation, we need to identify the resource which can be required for the requests of customers. We assume there are two types of resources, CPU and network bandwidth. The contract consists of m levels of QOS, each consisting of B_i^C, B_i^N , C_i and N_i. It states that when the client is being served a QOS of level i, it will require not more than C_i CPU cycles, N_i bytes of network traffic in every T seconds. To account for the client's burstiness, surplus quota is maintained up to a certain maximum limit(B_i^C for CPU, and B_i^N for network traffic) which can be used to serve the burst of requests. Similar to [AS98], there is also a reward R_i for each level based on the consumer perceived utility of the level. We use server admission control mechanisms very similar to those of [AS98].

4.1 Admission Control

The system keeps track of the average CPU $\bar{C}(i)$ and bandwidth $\bar{N}(i)$ utilization for a request of each i using a profiling mechanism. When a new request arrives, admission control decisions are taken based on the $\bar{C}(i)$ and $\bar{N}(i)$ assuming that this request will consume an average amount of the resources.

When a request comes which belongs to class i, it is admitted only if
$\bar{C}(i) \ < \ C(i)$, and
$\bar{N}(i) \ < \ N(i)$, and
In case it is accepted, $C(i)$, and $N(i)$ are decremented by the CPU, and network bandwidth consumption of the request respectively.

After every request, average resource utilizations($\bar{C}(i)$, and $\bar{N}(i)$) are updated based on the consumption of last request ($c(i)$, and $n(i)$) as follows :-
$\bar{C}(i) = \alpha\,\bar{C}(i) + (1-\alpha)\,c(i)$
$\bar{N}(i) = \alpha\,\bar{N}(i) + (1-\alpha)\,n(i)$
The parameters t and α will have to be tuned to give optimal results.

4.2 QOS Levels

The profiling system also keeps track of the system load state, and makes incremental adjustments in the customer QOS levels as in [AS98]. Periodic checks on system load state are performed, and when the system is overloaded, QOS level of some client(the one whose promotion yields the maximum profit) is promoted, and when it is under-loaded the QOS level of some client is degraded.

5 Implementation Platform

Each client has a web server running for it, which serves its requests(cgi scripts). All servers are different processes created using fork. The web server waits on some semaphores before actually running the cgi scripts,. A monitoring module runs as a separate process, which sets, or unsets the semaphores of each client based on the profiled behaviour of the client. By controlling the semaphores, the monitoring module controls the resource allocation to the web server processes corresponding to each client. As an example, there are 2 semaphores(sem0, and sem1) for each client. The web server waits on sem0 before allocating CPU(computation work), and on sem1 before allocating network bandwidth(actually sending the computed result). Any policy can be implemented in the monitoring module, which runs independently of the web servers. So to change the policy, we do not need to change the web servers.

The profiling system used is system process accounting. It has a very low overhead, and is accurate enough for our purposes. The monitoring module polls it periodically, and updates its estimates of resource consumption by each client. Based on this estimate, the monitoring module can set, or unset the semaphores corresponding to a client according to the monitoring policy it uses. Following is the description of the implementation of the policy described in section 4.

5.1 The policy tested

The resource consumption since last poll of a client is polled from the profiling system. If consumption of any resource by a client exceeds the quota allocated to the client, its semaphore is set. This means that the next request from this client will have to wait for the semaphore to get unset. It can only be done by the monitoring module, when the client has accumulated enough quota.

If the system is idle enough(measured by sum of resource consumptions by all clients), QOS level of a client is upgraded. The client chosen for this upgradation is one whose upgradation yields the maximum profit. Also, if the system is very busy, the QOS level of some client is downgraded(the one whose downgradation incurs the least loss).

The maximum delay which can be incurred to an incoming request will be in the case when all the clients have their maximum surplus quotas, and all clients send requests equivalent to(or more than) the quota allocated to them(both CPU, and network). In this worst case, delay is given by

$$D = maximum(\frac{\sum_{i=0}^{N} max_Cpu_Quota_i}{Cpu_capacity}, \frac{\sum_{i=0}^{N} max_Net_Quota_i}{Net_bandwidth})$$

Server level admission control is performed on this principle. If the admission of a new server increases D beyond what is tolerable to any of the clients, the server is rejected else it is accepted.

5.2 Testing procedure

The test computational grid was run, and a load generator played the part of customers. Each server has Poisson distributed burst arrival times, and each burst has an exponential number of requests sent back-to-back. The performance was measured in terms of response times, and throughput(number of requests served per unit time).

6 Testing and results

6.1 Testing procedure

The test computational grid was run, and a load generator played the part of customers. Each server has Poisson distributed burst arrival times, and each burst has an exponential number of requests sent back-to-back. The performance was measured in terms of response times, and throughput(number of requests served per unit time). The α's of servers, inter-updation-times of resource consumption etc. variables were varied, and results follow.

6.2 A low load test

3 clients were run on one machine, running 3 mini web servers, one for each clients. Requests were sent in bursts which contained a random(exponentially distributed with an average of 2 requests per burst) number of requests. The time between bursts was random(exponentially distributed, with an average of 800 milliseconds). All the involved clients had identical contracts with the grid.

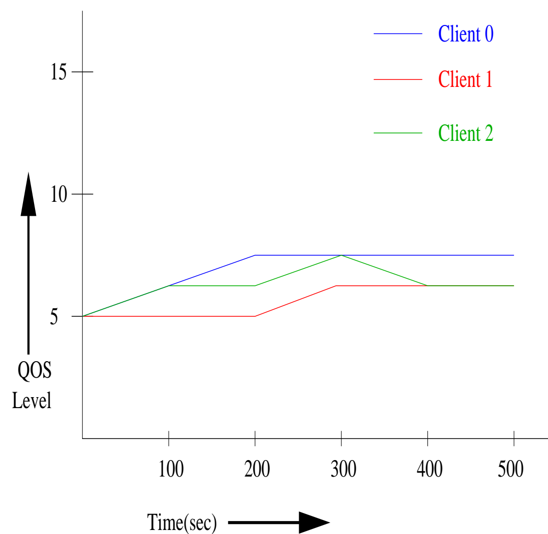


Figure 4: The low load test

The results of this experiment are depicted in figure 4. It can be noted that the QOS level of the clients did not vary much, but increased very slowly.

Client	Response time(ms)	Throuhput(req/sec)
0	458	1.0
1	388	1.2
2	366	0.9

Table 4: Throughput and response times in low load experiment

6.3 Increasing the load

Other details being kept the same, 3 additional client processes were added one for each of the same servers. The new clients had an inter-burst time of only 100ms. This resulted in increased load on the grid and hence the higher response times and lower throughputs. The QOS levels became unstable, but mostly varied between 4 to 8.

Client	Response time(ms)	Throuhput(req/sec)
0	2118	0.63
1	2287	0.58
2	1877	0.72

Table 5: Throughput and response times after increasing load

7 Summary

We studied some of the ways to solve the tricky resource allocation / admission control problem in computational grids, which is further complicated by the requirement of bundling of resources, and difficulty in quantifying QOS. The economic models compared it to free markets, and proposed a resource allocation strategy based on demand and supply. Going further in this comparison was the flexible QOS method, where resources were allocated to the customer when available, but kept the 'self-interest' of the server in mind. The admission control problem was not addressed directly in any of the above.

We propose a simple solution very similar to the flexible QOS method, and address the admission control problem by simply assuming that the incoming request will consume an average amount of resources, and later update the average.

8 Appendix : Implementation details

The code is has two main parts, the monitoring unit and the server. The terms which have relevance to the source code, have been enclosed in square brackets []. All the configuration files have their paths defined in the file code/ResourceControl.c.

8.1 Server

This part consists of a mini web-server for each client. For this, it forks into [NumClients] processes, and each process then runs a seperate web server on a different port. Presently, the ith web server runs on port [basePort] + i. Each process calls the function [proxify] to run its web server.

Each mini-web-server then waits for a cgi request(sample html form is in the file code/form0.html). Before processing the request, it waits on some semaphores, which allows the client to acquire the services of the grid only when/if the client deserves the resources. The present cgi scripts are all identical pieces of code(see code/Job0.pl), which calculate factorial of all numbers between [num1] and [num2]. The web server returns these factorials as reply to the incoming cgi requests as and when the semaphores allow it.

8.2 Monitoring module

The parent process of all the web-server processes runs the function monitorProcesses, which is the Monitoring module. This module implements the policy, which determines when/whether a client deserves resources to be allocated to it. The semaphores which the web-servers wait on are set/unset by this module at the appropriate time. After every [timeBetweenCondRefresh] seconds, the resource consumption by various clients are examined and based on this information the semaphores are adjusted as per the policy. It reads [policyIndepFile] and [policyDepFile] for configuration variables, whose formats are described in comments in the file code/ResourceControl.c.

8.3 Client

The file code/generateLoad.c is the source code for a client(end-user). It sends cgi requests in bursts on a specified IP address and port number. Burst size and time between different bursts are exponentially distributed random variables. It can be used to test the performance of the grid. More than one instance of it can be run at a time for stress testing. This program measures throughput and response time of the complete session, and prints them verbosely after the completion of every request.

8.4 Executing

Various file names(string constants in file code/ResourceControl.c) will have to adjusted before compilation. Simple "make" command compiles both the files ResourceControl.c and generateLoad.c into executables ResourceControl and generateLoad respectively.

Running ResourceControl will read configuration data from [policyDepFile] and [policyIndepFile], and run [NumClients] number of servers at specified port numbers. This file will also run the monitoring module to manage the incoming requests for the servers. This will need system accounting to be switched on(see file code/acctref). After this, one or more instances of client program(generateLoad) can be run and performance metrics be measured.

Acknowledgments

I would like to thank my guide, Prof Varsha Apte for the constant motivation and fresh ideas she provided. Her friendly disposition and invaluable support and encouragement have proved to be very rewarding.

References

[AS98] Tarek Abdelzaher and Kang G. Shin. End-host architecture for qos-adaptive communication. In *Proceedings of the Fourth IEEE Real-Time Technology and Applications Symposium*, pages 121–130, 1998.

[KS02] M. Toulouse K. Subramoniam, M. Maheswaran. Towards a micro-economic model for resource allocation in grid computing systems. In *IEEE Canadian Conference on Electrical and Computer Engineering*, pages 782 – 785, 2002.

[RWB00] J. Brevik R. Wolski, J. Plank and T. Bryan. G-commerce market formulations controlling resource allocation on the computational grid. In *Technical Report UT-CS-00-450, University of Tennessee*, 2000.

[RWB01] Todd Bryan Rich Wolski, James S. Plank and John Brevik. G-commerce: Market formulations controlling resource allocation on the computational grid. In *Proceedings of the 15th IEEE International Parallel and Distributed Processing Symposium Proceedings*, page 8 pp, 2001.

[TAB03] K.G. Shin T.F. Abdelzaher and N. Bhatti. User-level qos-adaptive resource management in server end-systems. *IEEE Transactions on Computers*, 52(5):678–685, 2003.