QoS-aware Middleware for Cluster-based Servers to Support Interactive and Resource-Adaptive Applications *

S. Senapathi B. Chandrasekaran D. Stredney† H. Shen D. K. Panda

Department of Computer and Information Sciences
The Ohio State University, Columbus, OH 43210
email:{senapath, chandrab, hwshen, panda}@cis.ohio-state.edu

† Ohio Supercomputer Center
Columbus, OH 43212
don@osc.edu

Abstract

Advances in commodity processor and network technologies have made cluster-based servers very attractive for supporting a large number of interactive applications (such as visualization and data mining) in the domains of Grid Computing and Distributed Computing. These applications involve accesses to huge amounts of data within the servers and heavy computations on the accessed data before sending out the results to the clients. The interactive *nature of these applications requires some kind of QoS support (such as guarantees on response time) from the underlying server. Unfortunately, the current generation cluster-based servers with the popular interconnects (Gigabit Ethernet, Myrinet, or Quadrics) do not provide any kinds of QoS support. Fortunately, many of these applications are* resource-adaptive*, i.e., application parameters can be changed to suit user demands and available system resources. To solve these problems, a new QoS-aware middleware layer is proposed in this paper for cluster-based servers with Myrinet interconnect. The middleware is built on top of a simple NIC-based rate control scheme that provides proportional bandwidth allocation. Three major components of the middleware (profiler, QoS translator, and resource allocator), their functionalities, designs, and the associated algorithms are presented. These components work together to execute a requested job in a* predictable *manner with an efficient allocation of system resources while exploiting the resource-adaptive property of the application. The complete middleware is designed, developed, and implemented on a Myrinet cluster. It is evaluated for two visualization applications: polygon rendering and ray-tracing. Experimental evaluations demonstrate that the proposed QoS framework enables multiple interactive and resource-adaptive applications to be executed in a predictable manner while keeping the allocation of system resources efficient. It is shown that the QoS-aware middleware helps applications to obtain response times within 7% of the expected times, compared to increases of up to 117% in the absence of any QoS support.*

*This research is supported by ITEC grant and NSF grants #CCR-0204429 and #EIA-9986052.

1 Introduction

Rapid advances in modern networking and commodity high performance systems are leading the field of computing towards the domain of Grid and Distributed Computing. This paradigm allows a large number of computers to be connected together through a high-speed network such as Myrinet[2], Gigabit Ethernet[3], or InfiniBand[4] to work like a single supercomputer. Such clusters are becoming increasingly popular for providing cost-effective and affordable computing environments for day-to-day computational needs of a wide-range of applications.

Traditionally, applications targeted for clusters have primarily included compute-intensive jobs. A new generation of applications is now being targeted for clusters such as data mining, imaging, collaborative interactions, virtual reality, multimedia server, web server, distributed visualization, collaborative computing, and tele-medicine [1]. Such applications possess two important properties of *interactivity* and *resource adaptivity*. Being interactive, these applications require a time limit on their execution times, which consequently poses a need for QoS guarantees that cannot be satisfied on current generation clusters. Due to this property of interactivity, such applications also require *predictable* execution time, where the client should be given a guarantee that the execution will terminate within a certain time-limit. These applications are also *resource-adaptive* and have parameters that can be assigned to a range of values based on the resource availability in the system.

Consider the example of an interactive remote visualization application shown in Figure 1. For such an application, a client typically needs to access data from a local file system associated with the cluster (or from a remote large scale data repository), perform computation on this data on the cluster so that the data can be rendered, and finally transmit the data from all the compute nodes to a front-end node, where the image can be viewed by the client.

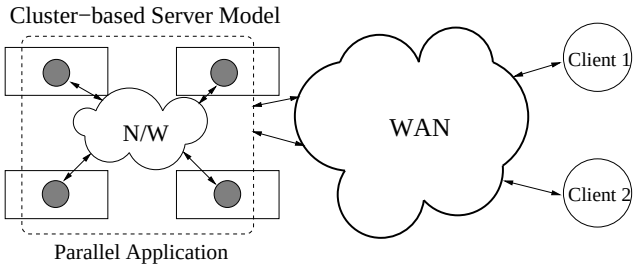


Figure 1: Client applications accessing a cluster-based server

Currently clusters that are used in a shared manner by such interactive applications have no means of guaranteeing performance demands in the face of network contention from other applications. Though CPU resources can be shared between contending applications using various schedulers available on the market, there is no similar scheme for sharing network resources. The challenge therefore, is to design a suitable QoS-aware middleware layer for clusters that can support the simultaneous execution of next-generation applications on shared clusters while providing the predictable execution time of each application. The requirements for such a middleware layer are enumerated as follows.

1. The framework must be able to make a translation from given application parameters to system resources.
2. The resource-adaptive property of applications can be used to determine the set of application parameters most suited to the available system resources, when conditions for parameters are not stated by the application.
3. Once a translation has been made, there must be a system-level mechanism that can provide coordinated access to system resources such as processing power and network bandwidth.
4. The framework must be able to admit as many client requests to a shared cluster as possible while taking into account their QoS constraints, and be able to execute the jobs while delivering the QoS requests of the admitted applications.

In this paper, we take on such a challenge. We present the design and implementation details of a QoS-aware middleware layer that satisfies the requirements given above. Implementation of the framework on Myrinet-based clusters, and the associated challenges are described. We also introduce new metrics to evaluate the performance of our framework. Experimental results on a 16 node cluster indicate that the framework is capable of providing application execution times that are not more that 7% higher than the expected execution times. It is also observed that in the absence of the middleware layer, execution times can go as much as 117% higher than the expected execution times.

The rest of this paper is organized in the following manner. Section 2 gives a high-level overview of the proposed middleware layer and its components. Section 3 explains the profiler module and its characteristics. Sections 4 and 5 describe the QoS translator and resource allocator modules. Section 6 further illustrates the working of the middleware layer through an example. Sections 7 and 8 present a definition of the metrics and a detailed performance analysis of the middleware layer based on these metrics. Section 9 outlines the related work in the field. Conclusions and future work are presented in Section 10.

2 Overall Structure of the Proposed Middleware Layer

Figure 2 shows the main components of the middleware: request handler, QoS translator, resource allocator, and profiler.

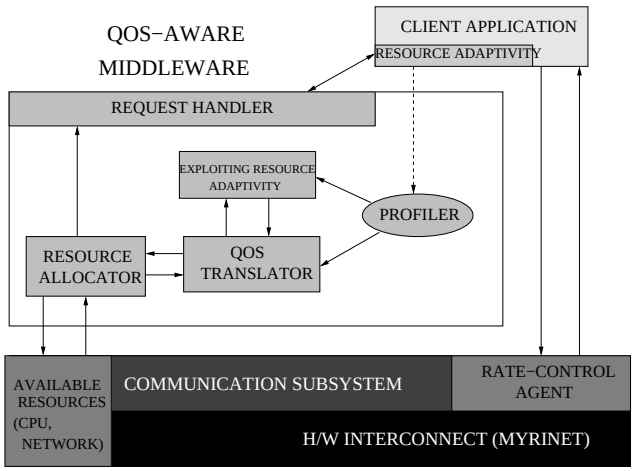


Figure 2: High-level overview of the proposed middleware layer

The profiler maintains profiled data of an application which characterizes application execution time with respect to different application parameters and system parameters. The QoS translator makes use of the profiled data to make the best possible match between application-given parameters and system resources and translates the given application parameters into system resource requirements. The decisions regarding actual allocation and de-allocation of the translated resource requirements are made by the resource allocator, based on the amount of resources available in the system. The request handler interacts with client applications and reads incoming application requests and extracts application parameter values from them.

The basic QoS mechanism is provided by a NIC-based rate control mechanism [5, 6] that controls the rate at which packets belonging to different flows are injected into the network based on the bandwidth assigned to those flows. Such a rate-control scheme can be implemented at the host, but a NIC-based scheme is preferred since it allows a finer granularity of control, because the NIC deals with frames whereas the host will deal with messages. The rate control mechanism was implemented in the GM messaging layer [9] over

Myrinet networks [2]. Application-level support was also added at the Message Passing level [10]. The next three sections explain the components of the proposed framework in detail.

Formal Definition: The formal definition of the input and output values of the middleware layer is as follows:
Input to the QoS-aware middleware: X ϵ Power Set$\{t_1, t_2, x_1, x_2 \ldots x_n\}$ where $\{x_1, x_2 \ldots x_n\}$ refers to the application parameters (such as image size, image resolution, etc.) and $\{t_1, t_2\}$ refers to the lower and upper time bound given by the application, respectively.
Output from the middleware: SUCCESS or FAILURE depending on whether the request can be satisfied.

Table 1 describes the working of the middleware layer. Here Y = {BW, N, t}, BW = bandwidth, N = number of nodes, and t = execution time.

```
Schedule (X)
1.      Y = QoS_Translate (X)
2.      if Not_equal (Y.BW, -1)
3.              Resource_Allocate (Y)
4.              Return SUCCESS
5.      end if
6.      Return FAILURE
```

Table 1: Formal Algorithm for the Middleware Layer

3 Profiler

The QoS translator needs profiled data on an application to determine the set of system resources required to satisfy user-given application parameters. The profiled data is obtained by executing the application on the cluster for different user parameters and system parameters (such as number of nodes and reserved bandwidth). The bandwidth allocation is achieved by the NIC-based rate control mechanism. By providing bandwidth allocation, the application seems to execute in a *virtual network* which is reserved exclusively for that application. Hence the application's communication can progress with no interference from other applications contending for the network. Here we are assuming that the CPU is allocated exclusively to that application throughout its execution (for both computation and communication). Since our system uses a User-level communication protocol, it puts very little load on the CPU for communication. Explicit CPU scheduling schemes can also be incorporated in our framework.

Currently the profiled data is static i.e., no learning mechanism has been incorporated. However this profiling mechanism can be made dynamic so as to reflect the changes in the cluster with little impact on performance.

Let us illustrate the functionality of the module using working examples of two visualization applications (as shown in Figure 3): Ray-tracing[8] and Polygon Rendering. The quality metrics for the two applications that are taken into consideration are image sizes and *interleaving factor* (IF). The profiling information for the polygon rendering application maintains the execution pattern of the application for given two image sizes of 512 and 1024, and a range of bandwidth assignments taken on 4, 8 and 16-node clusters. The profiled information for the ray-tracing application (Figure 3A) is similar but for an added parameter of the interleaving factor, which determines the quality of the final image. The x-axis measures the bandwidth assigned per flow, and the y-axis measures the time taken for execution, for a set of parameter values. When an application is executed on 4 nodes, and each of the nodes has communication flows to all the other nodes, on the single link from a node, there will be 3 outgoing and 3 incoming flows, making 6 flows in all. The bandwidth measured on a link is given by the total amount of bandwidth taken by all the communication flows on that link.

Resource Adaptivity: These graphs clearly show the resource-adaptive property of such applications, by which the application can execute with different user parameters based on the availability of system resources to achieve a certain response time. For example, for the ray-tracing applications, to achieve a response time of atmost 2 seconds for an image size of 512x512 with an interleaving factor of 0 requires atleast 2 MegaBytes per second (MBps) per flow on 4 nodes, and 1 MBps on 8 and 16 nodes. If the application gives hard conditions for the response time as 2 seconds or less, and for the interleaving factor as 0, and there is less than 15 MBps available on some of the 16 nodes of the cluster, using the above information, the framework can deduce that the only option is to execute the application with an image size of 512x512 on 4, 8 or 16 nodes. Without the use of this middleware layer, an application in the above conditions might execute with the larger image sizes, and suffer as a consequence, even though image size is apparently not a hard condition for the application.

Determination of knee point: Another point that can be noted from the graphs is that an application always does not require the maximum bandwidth available on a link. For all the curves shown, it can be seen that as the bandwidth assigned increases, the execution time drops drastically at first, but the curve flattens out as more bandwidth is given. For every graph, we define a *knee-point* value as the bandwidth value for which the overall execution time is greater than the minimum execution time by a small percentage value say, under 5 percent[1]. The profiler stores this knee-point value for all the graphs and the framework can allocate this value instead of the maximum bandwidth to achieve very close to peak performance.

Formal Definition: The structure of the profiled data can be defined formally as follows. The profiled data consists of a set of points, arranged into graph-curves for clarity. Let the graph-curves be denoted by $G_1, G_2 \ldots G_k$. Each graph-curve contains a set of

[1] Without loss of generality a 5% difference is used in this paper. It can be set to any other smaller value.

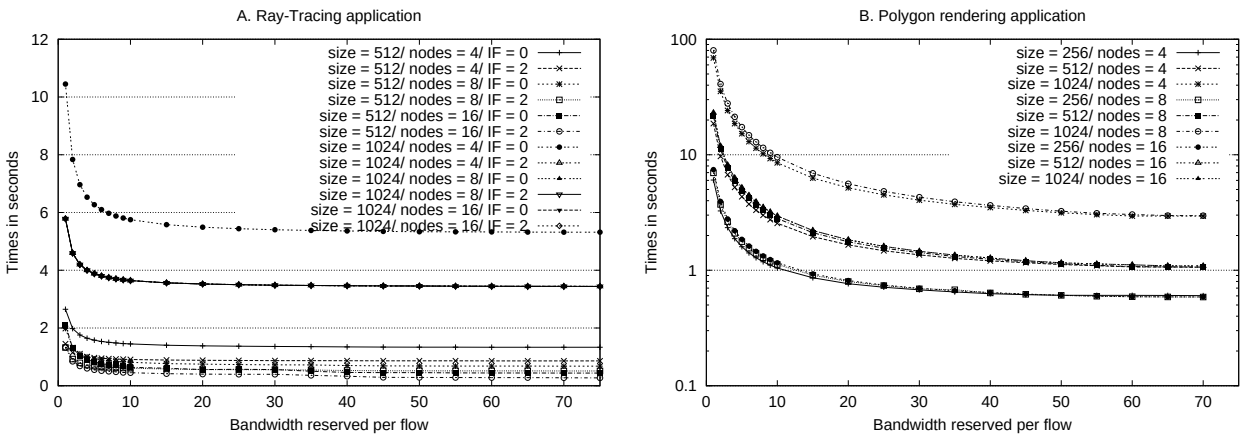


Figure 3: Profiling information maintained for the ray-tracing and polygon rendering applications with bandwidths denoted as reserved per flow

attribute values unique to that graph-curve, which can be used to identify it. These attributes are as follows: G_i.attr = $\{x_1, x_2 \ldots x_n, \mathrm{N}, \mathrm{K}\}$ where $\{x_1, x_2 \ldots x_n\}$ are the application parameters described, N = number of nodes, and K = Knee-point bandwidth of this graph-curve. Each graph-curve consists of a set of points: G_i.points = $\{P_1, P_2 \ldots P_L\}$ where P_i = {BW, t}, BW = Bandwidth, and t = execution time.

4 The Quality of Service Translator

The QoS Translator performs a search and optimization operation to obtain the set of system resources that most closely match the specified application parameters. The working of the QoS Translator is illustrated in the following section by iterating through the steps taken for a sample client request.

Specified Time Bounds: Consider the following example of a user request for the ray-tracing application. Let us assume that a client request requires the time to render a frame has to be less than 1 second. Figure 4a has the same data as the profiled data for application as shown Figure 3a, but shows only the pertinent graph curves, namely the lines that lie below the application given limit of one second. The shaded area shows the range specified by the application.

Specified Quality Metrics: Since the time constraint specified by the application is less than 1 second, and all graphs with points below 1 second have image size of 512x512, the image size that can be allocated to the application has to be 512. If the application has not specified the image size, or if the image size that it has given is equal to 512, then the translator goes ahead with trying to narrow down the set of points from which possible system allocations can be done. If the application has specified an image size and it is bigger than 512, then the translator deduces that this request cannot be satisfied, and returns the result to the request handler. For the current example, let us assume that the application desires an interleaving factor of 0 and has not specified any particular image size. Figure 4b shows the new set of points that satisfy the given criteria so far.

Knee Point Optimization: The knee-point value that was defined in the previous section can be incorporated into the equation. For the above example, Figure 4b shows the two graphs with the knee-point of each graph marked.

Once the translator has obtained the smallest set of points that can be used to satisfy the application request, it tries to find out if the corresponding system resources can be allocated by conferring with the resource allocator. The translator sends bandwidth requests to the allocator in increasing order, that is, the bandwidth corresponding to the least time value is sent first, and then for the next higher time value and so on. If the allocator finds an alternative that can be satisfied, it sends the result back to the request handler.

The translation works similarly for the polygon rendering applications, the only difference being the nature of the profiled data. For the polygon rendering applications, the only application parameter is the image size. Therefore the QoS translator narrows down the set of graphs to be considered using the time limit and image size, if given by the application.

Formal Algorithm: We now formalize the working of the QoS translator using the following algorithm:
Input to the QOS translator: X ϵ Power Set$\{t_1, t_2, x_1, x_2 \ldots x_n\}$.
Output from the QoS translator: Y_i = {BW, N, t}, BW = bandwidth, N = number of nodes, and t = execution time.
The point Y_i has final system resource allocation information to be sent to the resource allocator.

Table 2 shows the formal algorithm that describes the translation mechanism and the working of parallel search threads responsible for finding the set of valid points for each graph[2].

[2]The parallel search threads were implemented using the *pthread* mechanism on a single node.

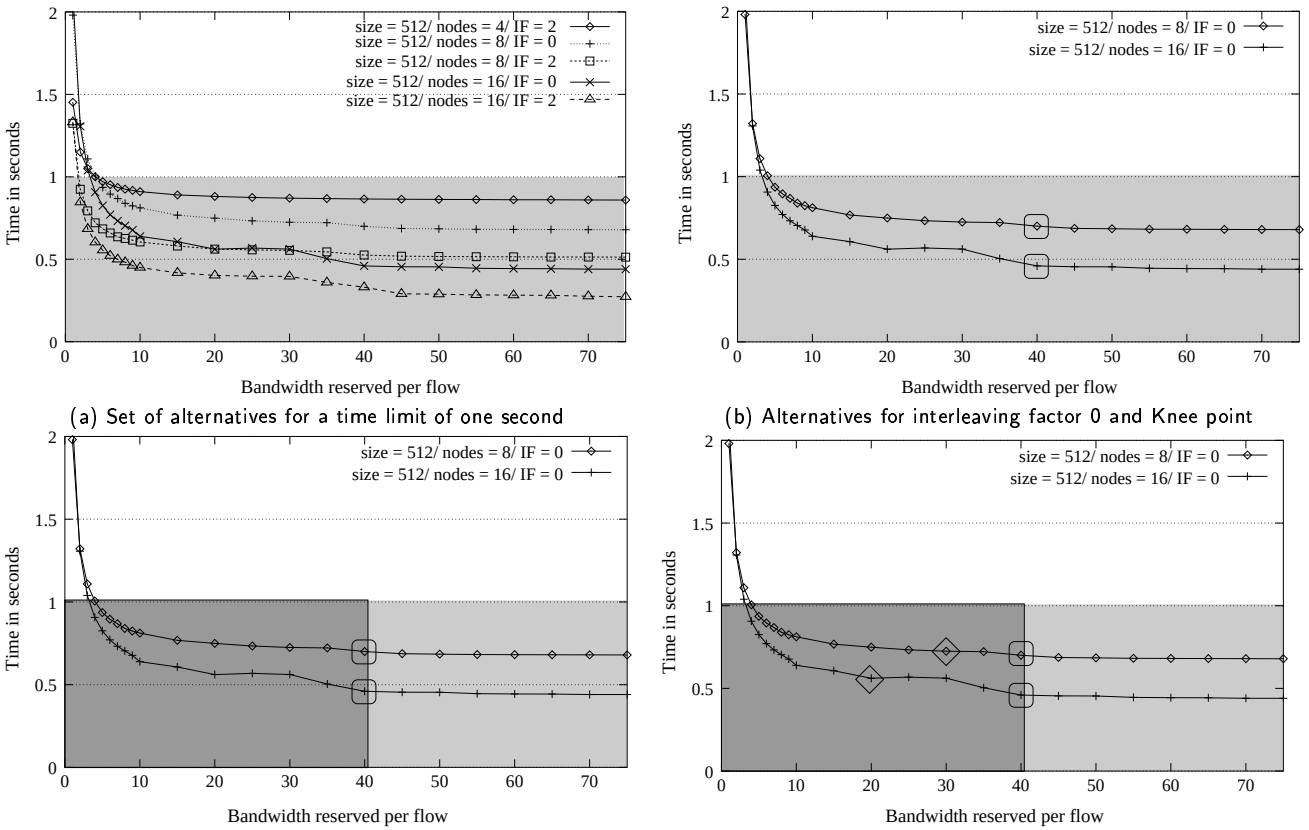


Figure 4: These graphs show the steps in the selection of points that satisfy given conditions for the chosen example

5 Resource Allocator

The selection of resources to be allocated depends on the number of processors available and the network resources available on the links of the cluster. The resource allocator has to use both the application-given criteria and the table of available resources to select the minimum set of resources that can satisfy the application's demands. The resource allocator maintains information about current reservations in the system such as CPU and network reservations. It also maintains a list of indices that denote the links with the highest available bandwidth so that allocations are spread evenly across all nodes in the system. As allocations and deallocations are made, this list is updated. The process of finding a suitable allocation is iterative in nature. The translator sends the alternatives to the resource allocator in order of best case to worst. The framework always tries to allocate resources for a user request so that its performance is maximized. The framework incorporates a greedy mechanism[3] of allocating resources which is augmented by careful use of the knee-point value. Instead of allocating all the bandwidth on a link to a requesting application, the resource allocator only needs to allocate the bandwidth corresponding to the knee-point, so that resources are being utilized efficiently. Figure 4c shows the incorporation of the knee-point values into the equation. For both the graph lines shown, the knee-point is at 40 MBps.

When the resource allocator is processing this request there can be three scenarios:

1. Even the smallest bandwidth of the shaded region cannot be satisfied. The resource allocator notifies the request handler about its decision, and the request handler in turn conveys the result to the application that it cannot be admitted.
2. For the selected set of curves atleast one of the knee points can be allocated. In order to maximize the performance of the application, the resource allocator tries to allocate the bandwidth that would give the highest time first, then the next highest time and so on. The search mechanism is done in parallel by a set of search threads described in Section 4. Once all the threads have completed their searching, the master thread compares the alternatives selected by each of the thread, and selects the best alternative that would provide the application with the least response time. In the above example, there will be two searching threads and the allocation of 40 MBps on 16 nodes which would give the best possible time for an image size of 512x512 and an interleav-

[3]For simplicity, we use a greedy algorithm in this paper. Any other algorithm with different cost complexity can also be used.

A. *QoS_Translate*(X)
1. for i = 1 to k
2. if match (G_i.attr, X)
3. Start_Thread (Search_Thread, X, G_i, i)
4. else
5. Y_i.BW = -1
6. end if
7. end for
8. Thread_Wait ()
9. y = Get_Smallest_Y ()
10. Return y

B. *Search_Thread*(X, G_i, index_of_thread)
1. for j = 1 to l
2. if match_time ($G_i.P_j$.t, X.t_1, X.t_2)
3. if allocate_possible ($G_i.P_j$.BW, G_i.N)
4. $Y_{index\_of\_thread}$.BW = $G_i.P_j$.BW
5. $Y_{index\_of\_thread}$.N = G_i.N
6. $Y_{index\_of\_thread}$.t = $G_i.P_j$.t
7. exit
8. end if
9. end if
10. end for
11. $Y_{index\_of\_thread}$.BW = -1

Table 2: Formal Algorithm for QoS Translation

ing factor of 0 is found by one thread. Therefore the resource allocator chooses to allocate 40 MBps per flow on 16 nodes for the requesting application.

3. Due to existing reservations, it may not be possible to allocate the knee-point value of any of the graph lines. If this is the case, each search thread returns the largest bandwidth value which can be allocated taking into account existing allocations and resources available. Figure 4d shows the scenario for the example that we had considered. The maximum bandwidth available is only 30 MBps per flow for 8 nodes, and 20 MBps per flow for 16 nodes. The resource allocator selects and allocates 20 MBps per flow, on 16 nodes, as shown in Figure 4d.

If a reservation can be made, the smallest Y value returned by *QoS_Translate*is sent to the resource allocator so that the information pertaining to the new allocation can be entered into the system. The resource allocator then makes the reservation call to the rate-control agents at the NICs of the nodes on which the reservation is made.

It is to be noted that the current framework assumes a perfect match between the profiled information and job requirements. However, the application input data and other parameters might vary. Our framework has potential to work with a reasonable variations of input parameters. We are exploring ways by which our scheme can be enhanced to handle wide variations. The current framework does not guarantee fairness. It can also be added to the scheduling scheme by taking into consideration an *age* metric for the tasks.

6 A Working Example of the QoS-aware Middleware Layer

Figure 5 shows the working of the middleware with a set of applications requests, allocations and de-allocations. The figure shows the progress of time on the x-axis in seconds. The upward arrows indicate the arrival of applications with reservation requests and their resource allocations, and the downward arrows indicate the termination of the executed applications, and the return of resources to the system pool of resources. In the figure IS and IF refer to the values for the image size and interleaving factor requested by the application, respectively.

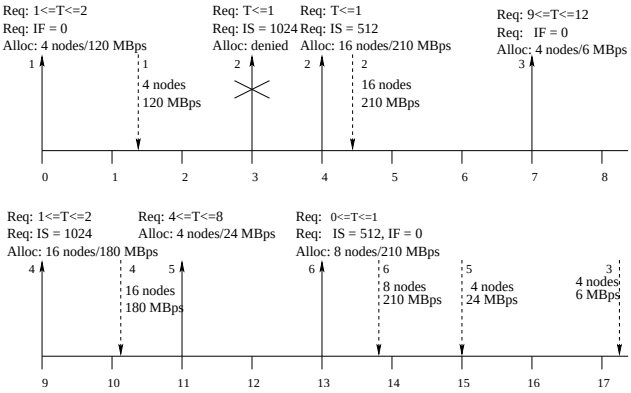


Figure 5: Working of the middleware layer with a set of resource allocations and de-allocations

1. Request 1 arrives at time t=0. It specifies that the execution time (T) be between 1 and 2 seconds, and interleaving factor of the image should be 0. The application does not pose any constraint on the image size. Since the graph with parameters of image size = 512/number of nodes = 4 nodes/interleaving factor = 0 has the lowest time, the translator picks that curve with knee point value 120 MBps. Thus, for Request 1, the middleware layer assigns 4 nodes with a bandwidth of 120 MBps reserved per bi-directional link. The execution terminates before any other requests arrive in the system.
2. Request 2 arrives at time t=3. It specifies that the execution time of the application must be below 1 seconds, and requires an image size of 1024. From the graph, it is clear that the system is incapable of delivering such a performance for the application. The framework therefore rejects this request.
3. Request 2 is re-submitted at time t=4 with the same time constraint, but with the image size decreased to 512. This request can be satisfied and from the profiled data the framework allocates 7MBps on 16 nodes.

The rest of the requests are satisfied in a similar way by the QoS framework. Thus, by the use of the profiling information stored about applications, and the resource-allocation matrix that stores system information, the middleware is able to allocate resources in such a way that application performance and resource utilization are both maximized.

7 Performance Metrics

To compare performance of applications when using the QoS-aware middleware layer with their performance when there is no such layer present, we define and use the following metrics:

1. We compare the execution times of applications obtained in the presence of the middleware layer with the execution times obtained when there is no such layer present. The comparison is done only for those jobs that have been *admitted* by the middleware layer.
   Percentage difference = $(T_{noqos} - T_{qos})/T_{qos}$, where T_{qos} = Execution time in the presence of the middleware layer, and T_{noqos} = Execution time in the absence of the middleware layer.
2. We also compare the execution times obtained with and without the middleware layer with the *expected* execution times of the application. Again, the comparison is done only for the *admitted* jobs.
   Percentage difference = $(T_{actual} - T_{requested})/T_{requested}$, where T_{actual} = Actual Execution time, and $T_{requested}$ = Requested response time of the application.
3. Finally we also show the admission rates for the middleware layer, that is, the percentage of incoming requests that are admitted.
   Admission rate = $N_{admitted}/N_{total}$, where $N_{admitted}$ is the number of jobs admitted by the middleware layer, and N_{total} is the total number of requests submitted to the framework.

8 Performance Evaluation

In this section, we present detailed, application-level evaluation to validate our framework as well as demonstrate its benefits.

Experimental Setup: In order to provide user requests that would be most similar to a real-world example, requests corresponding to visualization applications were submitted to the cluster-based server at random intervals. Also, when a user is rendering an image, he would try to look at the image from different directions, magnify a part of the image, or rotate the image, each of which will results in a further rendering action. To simulate this real-world scenario, the experimental setup consisted of a set of task requests arriving at random intervals, where each task is actually a further collection of subtask requests with different application level parameters and different random time intervals between them as shown in Figure 6. A subtask is a request for a single rendering application with specific application parameters. It must be noted that the CPU and network resources are allocated to a client for the full duration of a task request. This ensures that all the subtask requests that form part of a task can be completed once the task request is admitted into the system.

For such an experimental setup, we used the following parameters:

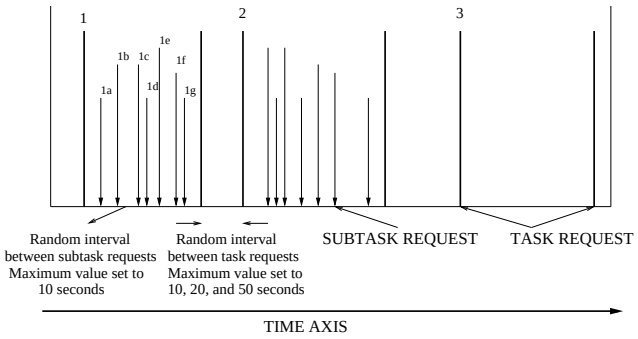


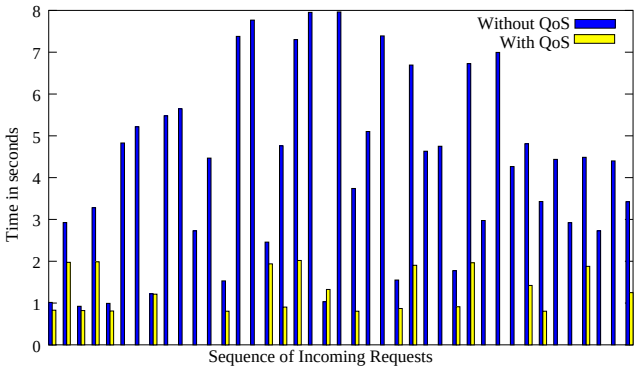
Figure 6: Experimental format showing the definition of task and subtask requests

1. The maximum time interval between successive high-level requests: This time interval was assigned values of 10, 20 and 50 seconds. The actual time interval between task requests was varied randomly between 0 and this maximum value.
2. The ratio of arrival image sizes: For the ray-tracing application, the image sizes of 512 and 1024 were varied in ratios of 50:50 and 30:70. The interleaving factors 0 and 2 were evenly distributed across the requests.
3. For the polygon rendering application, only the maximum interval time was varied.

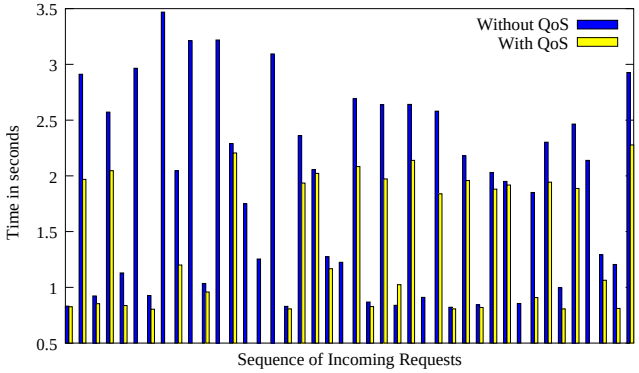
TestBeds: The experimental testbed consisted of a cluster of workstations with sixteen 1GHz Dual Pentium III processors, running Red Hat Linux kernel version 2.4.7-10 smp. These machines were connected by one 16-port Myrinet switches and LANai 7.2 NICs with 66 MHz processors. The communication layer running on the Myrinet cards was GM 1.5.1, and the MPI version was MPICH 1.2.1.7.

Application Results: We first evaluate the parameters T_{noqos} and T_{qos} described in Section 7 for the two applications presented. Figures 7 and 8 show the experimental results obtained for ray-tracing and polygon rendering applications. The x-axis shows the sequence of incoming requests. The y-axis shows the execution time for the application in seconds. The graphs are obtained by varying the maximum arrival interval time, and changing the ratio of user requests for image sizes of 512 and 1024. The light-colored bars on the graph show the execution times with the QoS middleware running on the system, and the shaded bars show the execution times when no such middleware is present.

Figure 7 shows the results obtained for the polygon rendering applications when the maximum inter-arrival time between requests is 10 and 20 seconds. The distribution of image sizes was random across requests, with the number of requests for each image size being approximately equal. Figure 8a shows the experimental results for ray-tracing applications when the ratio of image sizes of 512 to 1024 is 50:50 and the maximum inter-arrival time is 10 seconds as shown in the graph. Similarly the rest of figure 8 shows the results for the

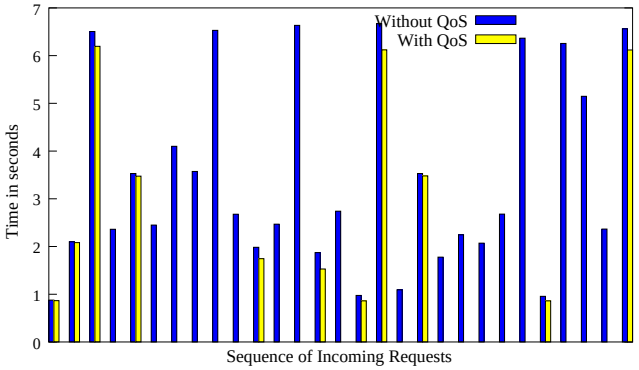


(a) Maximum interval = 10 sec

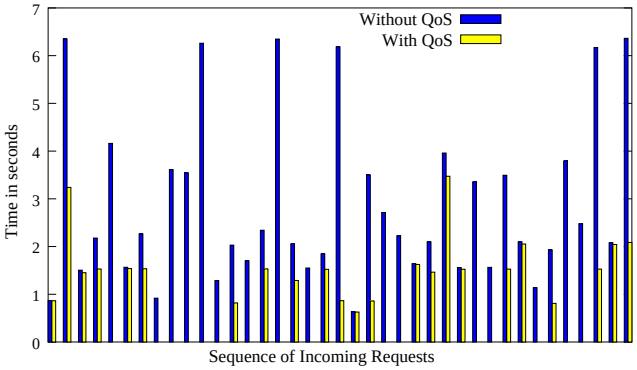


(b) Maxi-mun interval = 20 sec

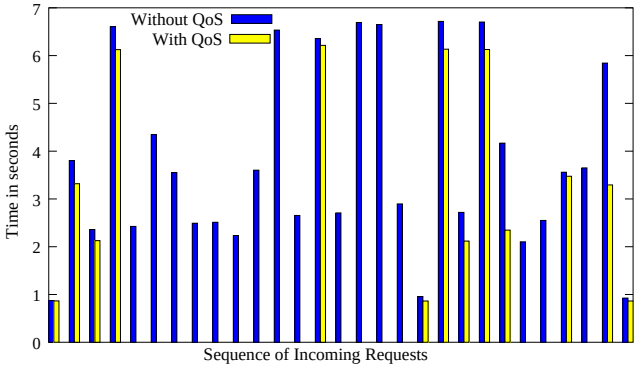
Figure 7: Experimental results for polygon rendering client applications with parameters as shown



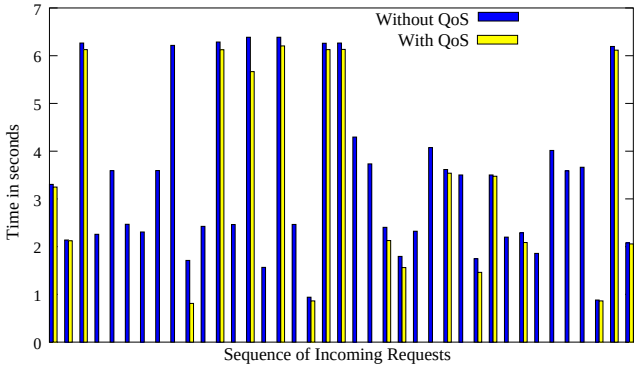
(a) 512:1024 = 50:50, Max interval = 10 sec



(b) 512:1024 = 50:50, Max interval = 20 sec

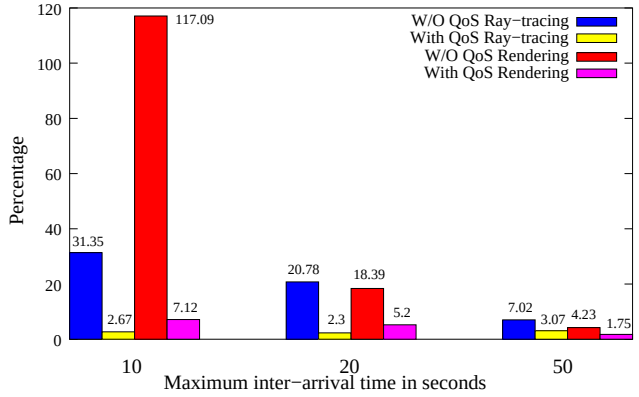


(c) 512:1024 = 30:70, Max interval = 10 sec

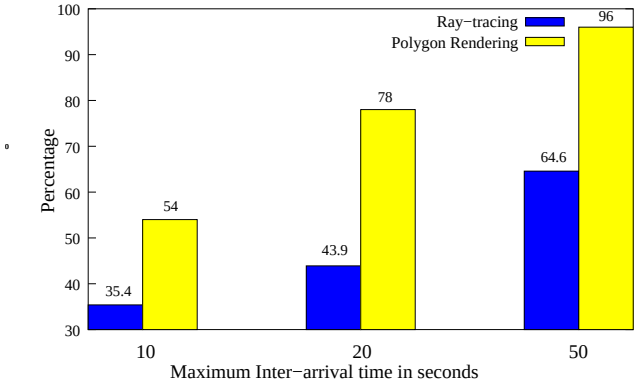


(d) 512:1024 = 30:70, Max interval = 20 sec

Figure 8: Experimental results for ray-tracing client applications with parameters as shown



(a) Percentage difference between expected and actual time



(b) Percentage of input request accepted by the middleware

Figure 9: Performance metrics calculated for the ray-tracing and polygon rendering applications

ray-tracing applications for the specified parameters. It can be seen that there are requests which execute when there is no middleware, but are not allowed to execute in the presence of the QoS framework. These requests cannot be satisfied due to low availability of system resources, and correspondingly, it can be seen that if these requests are allowed to execute, the execution times obtained are much higher than expected, and the user requirements are not satisfied. More results and analysis are available in [7].

Percentage improvement by the QoS middleware: This is the first metric discussed in Section 7 where we consider the performance improvement obtained by using the middleware layer. Up to 80% improvement can be obtained for the polygon rendering applications and up to 64% improvement can be obtained for the ray-tracing applications. The performance benefits gained are more for rendering applications because the communication to computation ratio is higher for these applications.

Percentage difference with the expected execution time: Here we try to compare the execution times obtained with and without the middleware layer with the expected execution times of the application. This is the second metric described in Section 7. Figure 9a shows the percentage difference between the expected time of execution given by a user request, and the actual execution time attained by the application. It can be seen that the difference in the percentage increase is most marked for the polygon rendering applications, again due to the fact that there is a high communication to computation ratio in these applications. This is as high as 117% for a maximum inter-arrival time of 10 seconds in the absence of our framework. As the inter-arrival times get larger, the contention and load in the system decreases, and so does the differences between the expected and actual times. It should also be noted that for the ray-tracing applications, though the communication contributes a very small percentage of the total execution time, there is a marked advantage in using the middleware layer due to the contention from other clients.

Admission Ratio of the Middleware Layer: Here we evaluate the third metric discussed in Section 7. Figure 9b shows the admission ratios of our framework under different load conditions for both the test applications. At heavy loads, the admission rate falls below 40% for the ray-tracing application, and below 60% for the polygon rendering applications. This lowered admission rate is validated by the lesser percentage increase observed in the previous graphs, showing that, by keeping the admission rate small, the system is able to guarantee the applications execution time that is close to the requested response time. As discussed in Section 5, a greedy allocation scheme is currently used in the resource allocator. The admission ratio can further be improved by implementing a different allocation scheme for the resource allocator. As the load on the system becomes lighter, the system is able to guarantee the applications the requested response time, while admitting more jobs, due to less contention in the system. At light loads, the admission rate reaches almost 100% for the polygon rendering applications. It should be noted that under light loads, using the QoS layer will add a slight overhead. These results are available in [7].

The results shown here demonstrate that the use of our proposed QoS-aware middleware layer guarantees to deliver close to the requested response time of client applications even in the presence of network contention from other applications. In the absence of such a framework, it can be seen that the resulting high load and contention in the system can lead to very high differences between the applications' requested execution time and the actual time the applications complete execution. This increase in response time at high loads is very undesirable for interactive applications. Since these are interactive applications, admitting a selected number of tasks for which response time is guaranteed is definitely better than admitting all tasks, which would result in undesirably high response time.

9 Related Work

The use of applications' resource adaptive nature to improve performance based on available resources has been studied extensively by a few other research groups in recent years. Darwin project [15] outlines a set of resource management mechanisms of *application-aware* networks to support QoS for applications. Chang et al [16] study the use of an application's tunability for efficient and predictable management of system resources. Using a *tunability interface* and *virtual execution environment* distributed applications can dynamically adapt to the network conditions. Zhang et al [11] explore the effect of using various scheduling mechanisms on the responsiveness and performance of applications. Chatterjee et al [12] describe models which facilitate adaptive QoS-driven resource management in distributed systems with heterogeneous components. The Globus project's GARA framework [13, 14] provides end-to-end QoS for different types of resources. Our approach is different from these approaches in the sense that our proposed framework uses application profiled data to match application parameters to required system resources. By this approach, the framework can determine the exact amount of system resources that can satisfy given application demands, therefore providing efficient utilization of cluster resources. Another point which makes our approach unique is the support provided at system level by a NIC-based rate control mechanism to control allocation of intra-cluster network bandwidths for application flows in a shared cluster.

10 Conclusions and Future Work

In this paper, we presented a QoS framework for supporting applications with resource adaptivity and predictable execution performance. The QoS-aware

middleware layer exploits the resource-adaptive property of applications to determine the exact resource requirements needed to satisfy application demands. In order to provide the best match between application parameters and system resources, we realized that the use of profiled data taken apriori about the application was the best option. The developed middleware layer uses the profiled data of applications to choose the best set of resources to satisfy an application's demands. Resource allocation is done in an efficient manner such that only the minimum set of system resources needed for the application to achieve its proposed performance goals are allocated. The framework is supported at the network level by a NIC-based rate control scheme that provides proportional bandwidth allocation to communication flows. The combined use of this rate control scheme and the middleware guarantees to first determine the exact resource requirements of applications and then to satisfy these QoS reservations for network and CPU resources. Therefore, the proposed framework promises to support next generation interactive applications (visualization, data mining, virtual reality, etc.) on a shared cluster. By using the proposed middleware, we were able to demonstrate that applications can obtain execution times within 7% of expected response times while in the absence of such a framework, they may encounter an increase of 117% over the expected response times.

Currently the framework only uses a proportional bandwidth allocation mechanism for reservation of network resources. We are exploring the integration of CPU and disk scheduling mechanisms with the middleware layer. The scheduling algorithm used by the resource allocator component of the middleware currently uses a greedy scheme in that it allocates a maximum set of resources to satisfy the needs of the applications. The effect of using different types of scheduling algorithms can be explored. It will be interesting to consider application classes with different priorities and re-evaluate the scheduling algorithms in this context. Currently the framework is implemented by a centralized scheme, where all requests are sent to a single central manager which then handles the requests and sends the replies back. Another possible future research direction can be the design of a distributed framework and the advantages and disadvantages of such a scheme can be compared versus the centralized scheme currently in practice. Currently the scheme is implemented and tested on Myrinet networks. Another direction of future research can be the implementation of the scheme on other high-performance networks, such as Gigabit Ethernet, and InfiniBand.

Acknowledgment: The authors would like to express their gratitude to Jinzhu Gao and Tony Garcia for their help with visualization applications.

References

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

[2] N. J. Boden, D. Cohen, et al. *Myrinet: A Gigabit-per-Second Local Area Network,* IEEE Micro, pages 29–35, Feb 1995.

[3] R. Sheifert, *Gigabit Ethernet,* Addison-Wesley, 1998.

[4] *InfiniBand Trade Association,* http://www.infinibandta.com.

[5] A. Gulati, D.K. Panda, P. Sadayappan, P. Wyckoff. *NIC-Based Rate Control for Proportional Bandwidth Allocation in Myrinet Clusters.* In Int'l Conf On Parallel Processing, September 2001.

[6] S. Senapathi, D.K. Panda, D. Stredney, H. Shen. *A QoS Framework for Clusters to Support Applications with Resource Adaptivity and Predictable Performance.* In Int'l Workshop on Quality of Service, May, 2002.

[7] S. Senapathi. *QoS-Aware Middleware to Support Interactive and Resource Adaptive Applications on Myrinet Clusters.* M.S Thesis, Dept. of Comp. and Info. Science, The Ohio State University, September 2002.

[8] A.Garcia, H. Shen. *An Interleaved Parallel Volume Renderer with PC-clusters.* Eugraphics Workshop on Parallel Graphics and Visualization, 2002.

[9] *The GM Message Passing System.*hhtp://www.myri.com/scs/index.html.

[10] W. Gropp, E. Lusk, N, Doss, N. Skjellum. *A High-Performance, Portable Implementation of the MPI, Message Passing Interface Standard.* Tech. Report, Argonne National Laboratory and Mississippi State University.

[11] Y. Zhang, A. Sivasubramaniam. *Scheduling Best-Effort and Real-Time Pipelined Applications on Time-Shared Clusters.* In Proceedings of the ACM Symposium on Parallel Algorithms and Architectures(SPAA), pages 209-218, July 2001.

[12] S.Chatterjee, J. Sydir, B. Sabata, T. Lawrence. *Modeling Applications for Adaptive QoS-based Resource Management.* In Proceedings of the 2nd IEEE High Assurance Systems Engineering Workshop, August, 1997.

[13] I. Foster, C. Kesselmann, C. Lee, B. Lindell, K. Nahrstedt, A. Roy. *A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation.* In 7th IEEE/IFIP Int'l Workshop on QoS, 1999.

[14] I. Foster, A. Roy, V. Sander, L. Winkler. *End-to-end Quality of Service for High-End Applications.* IEEE Journal on Selected Areas in Communications, Special Issue on QoS in the Internet, 1999.

[15] P. Steenkiste, A. Fisher, and H. Zhang, *Darwin: Resource Management for Application-Aware Networks*, CMU Technical Report CMU-CS-97-195, 1997.

[16] F. Chang and V. Karamcheti, *A Framework for Automatic Adaptation of Tunable Distributed Applications*, Cluster Computing: The Journal of Networks, Software and Applications, 2001.