Cluster Resource Management:

A Scalable Approach

Ning Li and Jordan Parker

Table of Contents

Table of Contents 2

Table of Figures 2

Abstract 3

1 Introduction 3

2 Related Work 4

3 A Scalable Approach – Hierarchy 4

3.1 Hierarchical management 4

3.2 Algorithms 5

3.3 Fault Tolerance 7

4 Results 8

4.1 Overview 8

4.2 Steady Workloads 8

4.3 Dynamic Workload 11

5 Conclusions 12

6 Future Work 12

7 References 13

Table of Figures

Figure 1: Hierarchal Management 5

Figure 2: Hierarchy Fault Tolerance 7

Figure 3: 4 Node Allocation Examples 8

Figure 4: 4 Nodes under a steady workload 9

Figure 5: 100 Nodes under a steady workload 10

Figure 6: 100 nodes with a 5x manager delay 10

Figure 7: 900 Nodes with a steady workload 11

Figure 8: 100 Nodes with a dynamic workload 12

Abstract

The last decade has seen an explosion in computing and in the latter half of the decade the Internet has brought these millions of computers together. With this explosion the performance of low cost personal computers has brought desktop computing closer in performance to server grade hardware. As this gap has narrowed the need for larger computing resources has been fueled by large demands on major web services. These two trends have led to a vast increase in very large clusters of commodity computers.

These new clusters with thousands of nodes have demonstrated high performance, scalability and fault tolerance thanks to the highly parallel nature of Internet workloads. As the popularity of these systems has grown it has become clear that there are needs for new resource management schemes. Significant work has been developed that makes single node resource allocation very successful, but managing many nodes has not yet reached any maturity. Much of the previous cluster resource management has depended on centralized managers, which we feel could be limiting factors in both scalability and fault tolerance for the largest clusters. Our hierarchal algorithm is able to achieve cluster wide usage ratios within 2% of our desired allocation with less than a 1% standard deviation. Beyond this more than reasonable performance our hierarchy should allow clusters to easily scale beyond a thousand nodes without management bottlenecks.

1 Introduction

Recently, the growth of very large clusters has exploded. These large clusters consisting of hundreds to thousands of nodes are becoming commonplace thanks to the growth seen by the Internet and the throughput required by the largest service providers. Additionally, the vast increases seen in the performance of PC microprocessors has made clustering more attractive. Clustering has a number of primary goals; it strives to achieve low cost performance ratios, high availability, and scalability. Clusters have been put to work in hosting among others the largest web sites and in computing large scientific simulations.

Ideally, a cluster is built from many low cost PCs and this can yield many advantages over multiprocessors. If a node is to fail hardware can be easily replaced often within days, and even the actual hardware can be obtained much easier than MP servers with can take months to acquire. Additionally, the buyer can easily upgrade the computing power by simply buying more nodes. Customers of large MPs must often buy the computing power they anticipate needing and can run into difficulties if they need more performance than their system provides. Clusters achieve all of this relatively easily, as they have inherent scalability and fault tolerance.

Unfortunately, MPs have one advantage over clusters in that they are a single system and managing a single system can often be much simpler than clusters, which have many completely separate nodes. Clusters are running many different classes of jobs concurrently. This means that some sort of scheme must be developed to make sure that each service class gets its fair share of resource pool, and balancing this resource pool among many separate nodes is far more difficult than on all but the largest MPs. Determining what is a fair share is one of the hardest questions here. For some clusters where users will place many long running jobs, the main goal is probably to provide long-term fairness. In contrast, an ISP hosting websites for a number of clients has probably sold some sort of bandwidth or resource guarantee. This means that if a class has an adequate workload to fill its share, it is entitled to that share at the minimum at any moment in time on the cluster. There are also many other reasons why a cluster operator might want some high-level resource management. One example are E-commerce customers who might want to give priority to certain customers and some sort of high-level scheme would be important to provide coordinate this scenario. The previous situation is beyond the scope of this paper as this line of research is in its infancy and some lower level advances still need to be made.

Much work on resource management for systems like NOW [3][4] and Condor [7] have been explored. These systems have very different workloads than short jobs seen by clusters serving web pages. The focus of our work will be on this later case in trying to provide a fair cluster wide resource allocation for systems such as web servers with many short requests. We hope prevent classes with large numbers of outstanding requests from flooding the systems and prevent poor or fragmented dispatching of jobs from degrading the overall desired share. There has been some other work in this specific area resource management. However, we believe that these other solutions though effective are not yet complete. Most of these schemes seem incomplete in their scalability. The bank algorithm we present has encouraging

results in both performance and scalability and might present a better option for very large clusters than previously researched management schemes. We found that we were able to sustain overall allocations within 2% of the desired load under adverse conditions with a standard deviation of less than 1% than only decreased as the clusters grew. Additionally, our algorithm is able to rapidly recover from allocation based on under usage through dynamic workloads.

2 Related Work

There has been much research into cluster management. Some early work ,such as that on the Condor project focused on long term fairness among user allocation[7]. The recent changes in the usage of clusters especially due to the highly parallel and fine-grained jobs on clusters have changed the focus of recent research. Research into very accurate single node resource management has been developed such as lottery scheduling resource containers and those methods are assumed in our research[10][5].

Armando Fox et al developed a prototype cluster that attempted to directly tackle the new problems and workloads that the web would provide[1]. They determined that there would be periods of both steady and bursty workloads. They also developed a number of concepts that have become widely applied in real systems. The concept of BASE and eventual consistency can be seen in many websites today. They also developed a cluster organization where front end nodes would receive jobs and with the help of the a single centralized manager would allocate those jobs for processing in back end worker nodes. Their management scheme seemed to prioritize load balancing and though fine for smaller single workload clusters, this sort of management might not be sufficient for very large clusters due to both scalability and the more diverse workloads that can be expected on the largest clusters.

Cluster Reserves were developed by Mohit Aron et al to help tackle the problems of the allocation of different job classes on clusters[2]. Cluster Reserves are an abstraction of a global resource principal for clusters. They formulated global resource management as an optimization problem. Their solution again depended on a single manager, which we believe with the load of solving large optimization problems and managing thousands of nodes could become a bottleneck.

Andrea Arpaci-Dusseau and David Culler implemented a global proportional share scheduler[3]. Their system extended stride scheduling and unlike the previous example had no central managers. Instead they had each node talk to some small and random set of other nodes to determine a desired local allocation. Their system suffers from not having been specifically designed for the short workloads that we propose and a lack of testing on larger clusters.

One of the major problems with the majority of this related work is in our view the scalability of their management, which is highlighted by presenting experiments with less than 50 nodes, most fewer than 10.

3 A Scalable Approach – Hierarchy

This section presents a scalable approach to achieve performance isolation between service classes hosted on a number of clusters. First, the hierarchical management structure is introduced. The hierarchical structure is vital in solving the scalability problem faced by a traditional centralized manager approach. Then, two algorithms -- cluster reserves and bank algorithm -- are presented. Either algorithm could be used by a manager to determine best resource allocation for each service class on each node that manager manages. Finally, the fault tolerance issue is addressed. These three parts together form our scalable approach to do clusters resource management.

3.1 Hierarchical management

As clusters are becoming increasingly large to the order of thousands of nodes, scalability is becoming more and more a problem for centralized manager approach to do cluster resource management. We believe hierarchical management is the solution to the problem and our idea is strongly supported by our results presented in next section. We use Figure 1 to illustrate the hierarchical structure we employ to do scalable resource management.

Only bottom level nodes actually service jobs. They are also named level_0 managers since they manage resource within itself. All managers of level_1 and above facilitate cluster resource management. Level_l+1 manager manages a group of level_l managers. Normally, two levels of management (not including level_0) are sufficient in most cluster resource management applications. One thing to note here is that here nodes and managers are only logically separated entities, they could very well physically reside on the same machine.

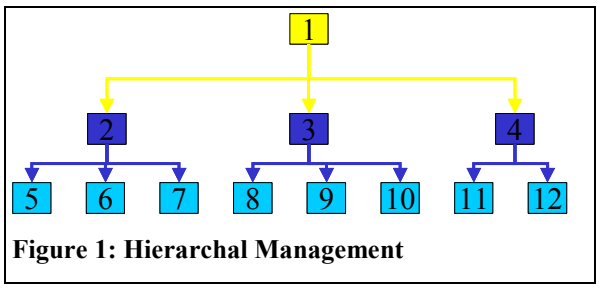

Figure 1: Hierarchal Management

Another good thing about hierarchical structure is that it is a natural fit for managing geographically distributed clusters that cooperatively provide the same set of services (e.g. web servers and their mirror ones). One centralized manager is impractical in this case since it will cause too much network traffic between the two distant clusters.

3.2 Algorithms

Two algorithms -- cluster reserves and bank algorithm -- are presented in the next two subsections. Either could be used by a manager to compute new resource allocation for each service class on each node the manager manages.

3.2.1 Cluster Reserves

M. Aron et al presented the cluster reserves algorithm in their paper [2]. We briefly introduce the cluster reserves algorithm here.

Goal: Let the cluster consist of N nodes and S service classes. Let r and u be NxS matrices such that r[i,j] and u[i,j] denote the percentage resource allocation and resource usage respectively at node i for service class j . Let D be a vector composed of S elements such that D[j] gives the desired percentage resource allocation for the cluster reserve corresponding to service class j. Given input matrices r and u and the vector D, the resource manager computes a NxS matrix R such that R[i,j] gives the new percentage resource allocation for service class j on node i.

The problem is formulated into two constrained optimization problems as follows:
Step 1: Compute the least feasible deviation between the desired and actual allocations.
The objective can be stated as:

*Minimize sum{j in 1..S} abs((sum{i in 1..N} R[i,j]) - N*D[j])* (1)

Additionally, the problem is constrained as follows:

for each node {i in 1..N}:

sum{j in 1..S} R[i,j] <= 100

for each service class on each node {i in 1..N, j in 1..S}:

R[i,j] <= u[i,j] if r[i,j] > u[i,j]

for each service class on each node {i in 1..N, j in 1..S}:

R[i,j] >= 1

Step 2: Compute the new resource allocations such that (1) the deviation V computed in the first step is achieved, and (2) the computed resource allocations are close to the service class usage on each node.
The objective can be stated as:

Minimize sum{i in 1..N} (sum{j in 1..S} (R[i,j] - (u[i,j] + k[i,j]))^2) (2)

And the problem is constrained as follows:

*sum{j in 1..S} abs((sum{i in 1..N} R[i,j]) - N*D[j]) = V*

The rest of the constraints are the same as in Step 1.

And k[i,j] is defined as:

*k[i,j] define= min(5, 500 * (D[j] - u[i,j]) / D[j]) if u[i,j] < D[j]*
*k[i,j] define= max(-5, 500 * (D[j] - u[i,j]) / D[j]) otherwise*

The resulting R[i,j]'s of matrix R after Step 2 are further processed in Step 3 to yield the new percentage resource allocations.

Step 3: Distribute unassigned cluster resources to idle service classes whose allocations fall below their desired cluster-wide allocation.

We first produce the problems in the AMPL modeling language, then feed it to the AMPL tool, which in turn uses LOQO to solve the constrained optimization problem. That's why all objectives and constraints above are written in AMPL-like style. However, 'abs' does cause trouble in AMPL and LOQO, and some modifications are necessary in order for AMPL and LOQO to solve problems correctly[6][9].

3.2.2 Bank Algorithm

Now we present our bank algorithm, which also computes a NxS matrix R whose R[i,j] gives the new resource allocation for service class j on node i. The advantages of the bank algorithm are, firstly, it is computational much simpler, thus much fast; secondly, it is much more flexible, so we could do different policy tunings in any of the 6 steps of the algorithm, to make the algorithm fit different goals.

Before we explain why we have those two advantages, we introduce a basic version of the bank algorithm (because of its flexibility, we could have many different versions of the algorithm with differently tuned policies in any of the 6 steps).

First, some terminology and primitives used:
Ticket: ticket represents resource rights
Bank account: bank accounts are set up by managers for each service class
Primitives: ticket transfer, ticket inflation/deflation, ticket deposit/withdraw

The ticket resource primitive is similar to that developed by Waldspurger and Weihl in Lottery Scheduling [10]. To be more consistent with the percentage allocation used in cluster reserves algorithm, there are exactly 100 tickets on each node and we compute the ticket allocation out of the 100 for each service class on each node.

Next, the 6 steps of the bank algorithm:
Step 1: for each service class on each node, deposit unused tickets
During previous round, any allocated but unused resources are proportioned amongst other service classes dynamically by the resource container mechanism. This contribution is not compensated in cluster reserves algorithm, but is compensated here in the bank algorithm.

Step 2: for each service class on each node, assign initial values for new allocation R[i,j]
$R[i,j] = u[i,j] + c$ if class j fully utilized previous allocation on node i
$R[i,j] = u[i,j] - c$ if class j under utilized previous allocation on node i
where c is a small number of tickets for adjusting R[i,j] toward best values
In case of full utilization, class j could potentially use more resource thus c tickets are added. In case of under utilization, class j would potentially use less resource thus c tickets are subtracted. In our experiment, a small adjustment number of 2 is used.

Step 3: for each service class, compare total allocation to its desired proportion,
subtract from the over-allocated and add to the needy and under-allocated
In this step, we are doing addition and subtraction here to bring the aggregate cluster-wide (here cluster refers to all nodes this manager manages) allocation to the desired allocation level. Needy means the class could take more tickets, i.e. it fully utilized its previous allocation.

Step 4: for each service class, withdraw tickets if still over-allocated,
and deposit tickets if still under-allocated
In this step, we withdraw over-allocated tickets from bank to make classes pay for what they take, and deposit under-allocated tickets to bank to compensate those that contribute.

Step 5: withdraw tickets and reward to the needy nodes
For those classes that have a positive number of tickets in their bank accounts, reward the tickets to the classes on needy nodes. Needy has the same meaning as in step 3.

Step 6: normalize allocation on each node and clear bank accounts for each service class
We normalize the allocation on each node so that the total number of tickets is 100. We also clear bank accounts for each service class when we do only spatial compensation. Spatial compensation will be defined in the next paragraph.

Two terms are very important for us to understand the second difference between cluster reserves algorithm and bank algorithm: spatial compensation and temporal compensation.

Spatial Compensation: If a class's desired allocation is t tickets, but it would only use t-s tickets thus only reserves this much on one node, it could be compensated by reserving t+s tickets on another node. This compensation of one class on a different node during the same round of allocation is called spatial compensation.

Temporal Compensation: If a class's desired allocation is t tickets, but it would only use t-s tickets thus only reserves this much on the node in this round, it could be compensated by reserving t+s tickets on this same node during next allocation. This compensation of one class on the same node (could also be on a different node) during a different round of allocation is call temporal compensation.

Now we are ready to revisit the two advantages of the bank algorithm. The first advantage of simplicity is relatively easy to see since cluster reserves algorithm involves solving constrained optimization problems while bank algorithm involves only pure arithmetic operations.

One example of bank algorithm's second advantage of flexibility is that the basic bank algorithm could be easily modified to achieve different levels of temporal compensation while cluster reserves algorithm only does spatial compensation. In step 1, bank algorithm compensates class's contribution of tickets in last round by depositing those tickets in its bank account in this round. In step 6, the basic bank algorithm clears bank accounts to limit the effect of temporal compensation in order to compare with cluster reserves algorithm. We could retain all account balance to do full temporal compensation, or retain a portion to seek a balance between no temporal compensation and full temporal compensation. It really depends on the goal: whether we want to achieve desired proportion during each round, or achieve desired proportion on average for a number of rounds.

Steps 2 through 5 of the bank algorithm are designed to achieve spatial compensation since spatial compensation are desired in almost all cases. Among them, Steps 3 and 5 play a relatively more important role in bringing aggregate cluster-wide allocation to the desired allocation level by rewarding to needy nodes of contributing classes. Due to time constraints, we didn't fully explore how to incorporate temporal compensation into cluster reserves algorithm. But we believe it could be done thus would provide cluster reserves algorithm with great flexibility.

3.3 Fault Tolerance

Though we will not present simulation of the potential usages of hierarchal fault tolerance we believe that it is an important aspect of clusters and their corresponding management schemes.

Here we introduce a manager replacement strategy to achieve fault tolerance in case of manager failure. When a level_l manager detects its level_l+1 manager's failure, it communicates with other level_l managers managed by its level_l+1 manager and they together decide which level_l manager becomes the new level_l+1 manager. If this manager does not become the new level_l+1 manager, it simply starts to

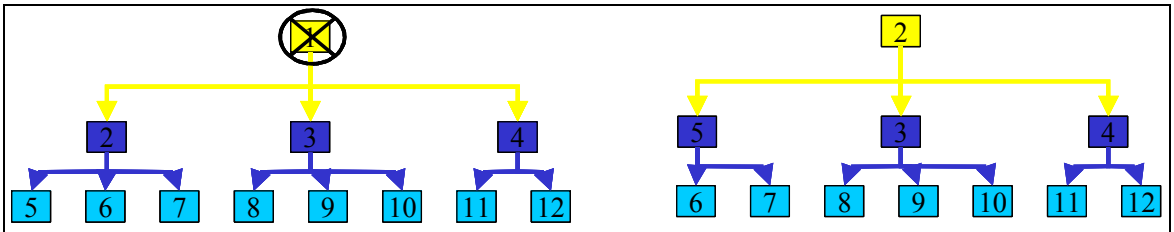

Figure 2: Hierarchy Fault Tolerance

report to the new level_l+1 manager. If this manager does become the new level_l+1 manager, it starts to report to the old level_l+1 manager's level_l+2 manager, if there is one; it also starts to manage those other level_l managers. Further, it notifies all level_l-1 managers it originally manages, and either designates one to replace itself or let all of them together decide who should become their new level_l manager and report to the new level_l+1 manager.

In Figure 2, we show a simple example that demonstrates this manager replacement strategy solves one manager failure problem. First Mgr1 dies. Mgr2, Mgr3, and Mgr4 detect Mgr1's failure, communicate with each other, and decide that Mgr2 will become the next level_2 manager, which is also the top manager. Then since Mgr3 and Mgr4 do not become the new level_2 manager, they'll start to report to Mgr2, which is the new level_2 manager. Now Mgr2 becomes the new level_2 manager. Since it's also the top manager, it does not have to report to anyone. It starts to manage Mgr3 and Mgr4. It also needs to notify Node5 Node6 and Node7, and could designate Node5 to replace itself. Finally, Node5 become a level_1 manager, manages Node6 and Node7, and starts (or rather, continues) to report to the new level_2 manager Mgr2. The result is shown on the right side of Figure 2.

4 Results

4.1 Overview

Our results were computed using NS, the network simulator [8]. The backend communication network was designed using simple 10Mbs links under UDP communication. We used UDP and this low bandwidth network to support our algorithm as being a viable low overhead management alternative. We succeed in this by having packets that are in our simulations no larger than 33 Bytes in length. The message length is (9 + 8 * number of service classes) bytes, making clusters with many service classes send messages still smaller than 1K. We record the number of tickets with integers, but if it were decreased to 1 or 2 bytes in size the message size could be further decreased. This results in our tests being able to send approximately 3000 messages over our low-bandwidth network per second, assuming that the entire backend shares the bandwidth on a single hub rather than a more typical switch based network.

The other important aspect of our results is that we assume ideal resource management on the individual nodes. Lastly, we had hoped to simulate clusters of sizes up to 10,000 nodes, but due to NS's very large memory requirements simulations were limited to around 1000 nodes on a system with GB's of local memory. The charts shown in this section demonstrate similar performance to those demonstrated in the cluster reserves [2]. Thus all of the data presented reflects our bank algorithm and the preceding description of cluster reserves should serve as a point of reference for alternative ways to examine the benefits of hierarchal management in clusters.

We are attempting to provide global resource allocation and this example will help to provide a clearer picture of our ideals and what we hope to avoid in our simulations. There are four nodes and the desired allocation for the overall cluster is 60% for class 1, 30% for class 2, and 10% for class 3; this is same allocation that is tested in our experimentation. In this example the dispatcher is only sending requests for class 3 to node 4. In static proportioning this would yield the 60/30/10 usage for the fourth node and leave the other nodes with 66/33/0 usages, and overall this would provide only 2.5% usage to class 3 as seen in the static management of Figure 1. Ideally, a global resource allocation could adapt to this dispatching and give the third class a 40% allocation and corresponding usage. This would yield an overall allocation of 60/30/10 as desired in the ideal management seen in Figure 3.

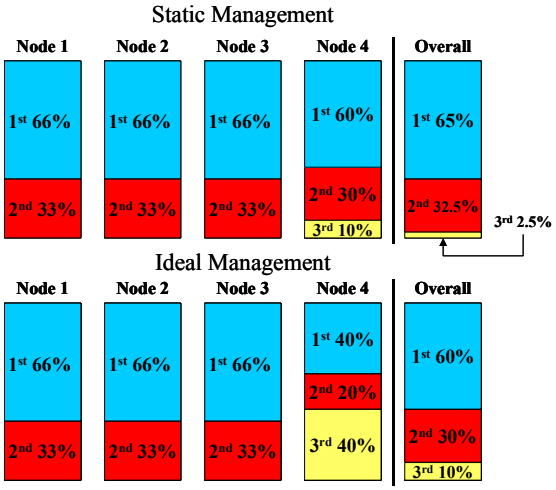

Figure 3: 4 Node Allocation Examples

4.2 Steady Workloads

The first test explores the simple example

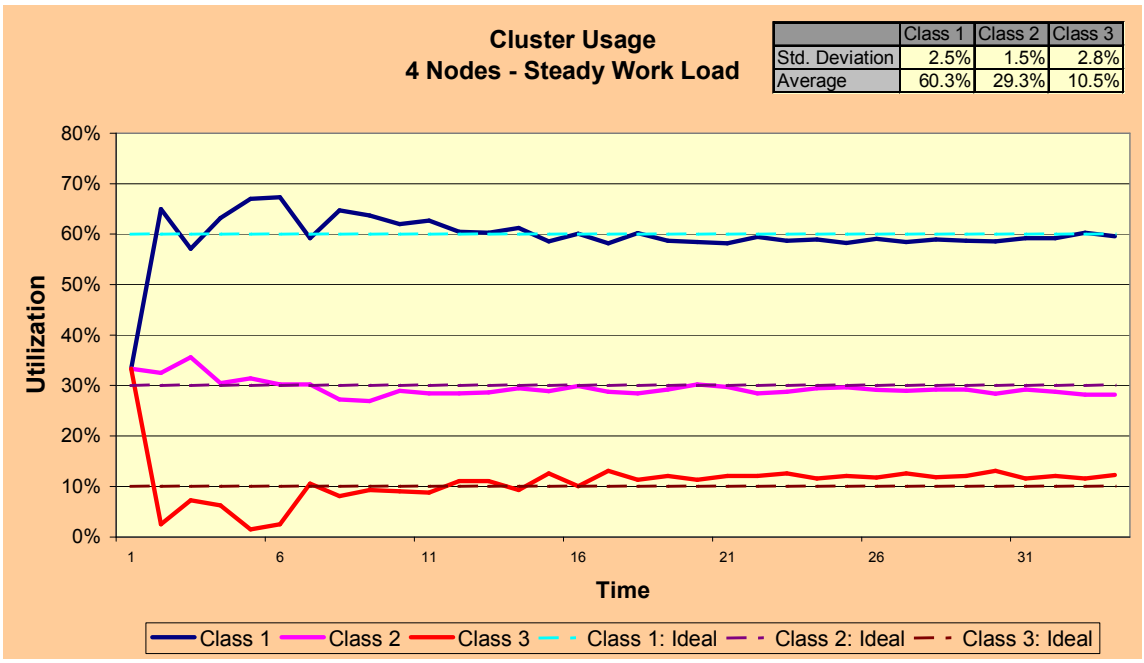

Figure 4: 4 Nodes under a steady workload

from Figure 3. This test involves our bank algorithm with a 2 level hierarchy and the same constraint of only sending requests for service class 3 to one of the four nodes. The results of this were fairly encouraging, after the startup the period there is an average allocation of 10.5% for the third class and the standard deviation is only 2.8% that can be seen by our fairly smooth series in Figure 4.

Next we increase the number of nodes to one hundred and also create a 2 level hierarchy where each node reports to one of ten managers and those managers all report to one top level manager. Additionally, all of the requests for the third class are sent to the first 30 nodes. The workload is unbalanced with far more than 10% of requests being issued on class 3 so as to place more pressure on the other classes, but since we have ideal local management this has little consequence. A reference set where our nodes had a fixed 60/30/10 management was also tested for this data, all of which can be seen in Figure 5. As more pressure has been placed on the managers to distribute the allocation for class 3 we see that some loss of the allocation for this class has occurred with an average of 8.9%. This is considered acceptable as the algorithm is designed such that all nodes in the cluster receive some piece of the allocation for the class in preparation for requests. Telling the algorithms and managers that the other nodes should not expect any requests defeats the intentions of this research. We can also see that with this larger cluster the allocation has become even steadier and we have standard deviation of only 0.58%.

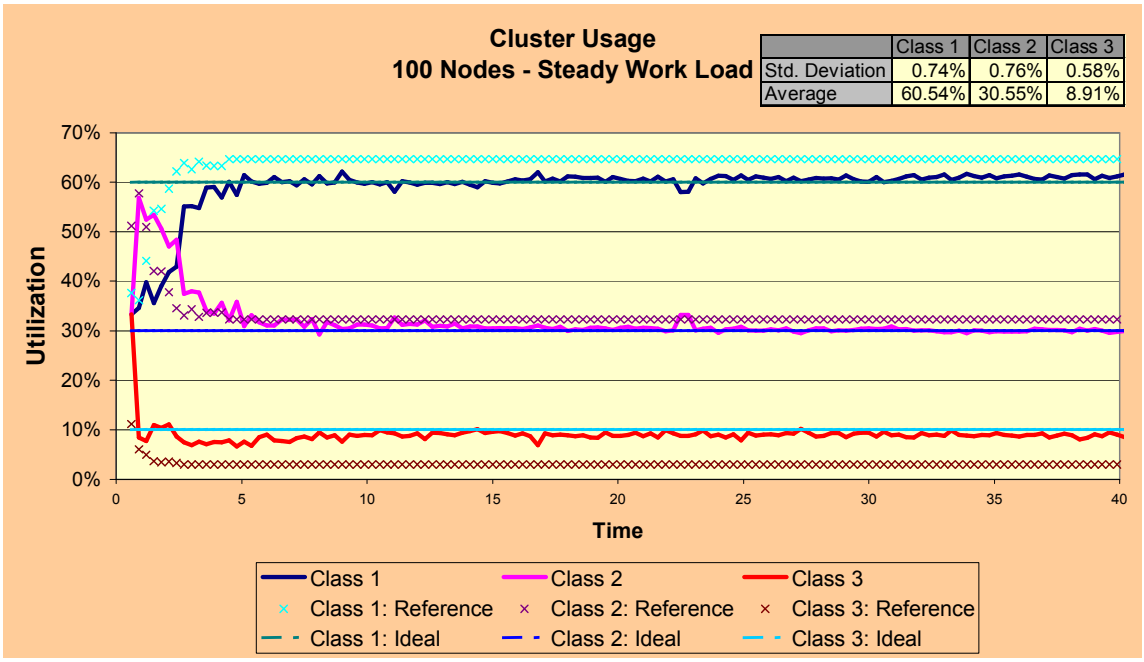

Figure 5: 100 Nodes under a steady workload

We also tested 100 nodes with a different reporting rate between nodes to managers and mangers to mangers. Nodes reported every 0.3 seconds and managers every 1.5 seconds. The results are very similar to that with more frequent reporting with somewhat slower changes, but a similar standard deviation and a marginally smaller average as seen in Figure 6. This test indicates that this algorithm might be suitable for geographically separate sub-clusters, which could be an attractive option for largest cluster operators.

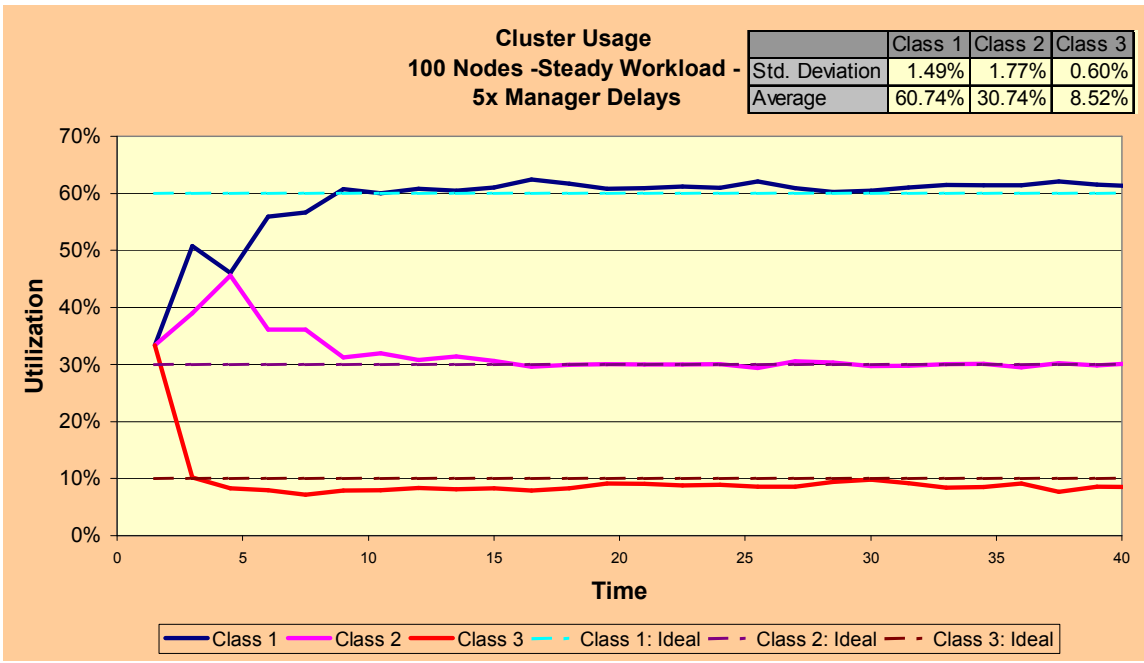

Figure 6: 100 nodes with a 5x manager delay

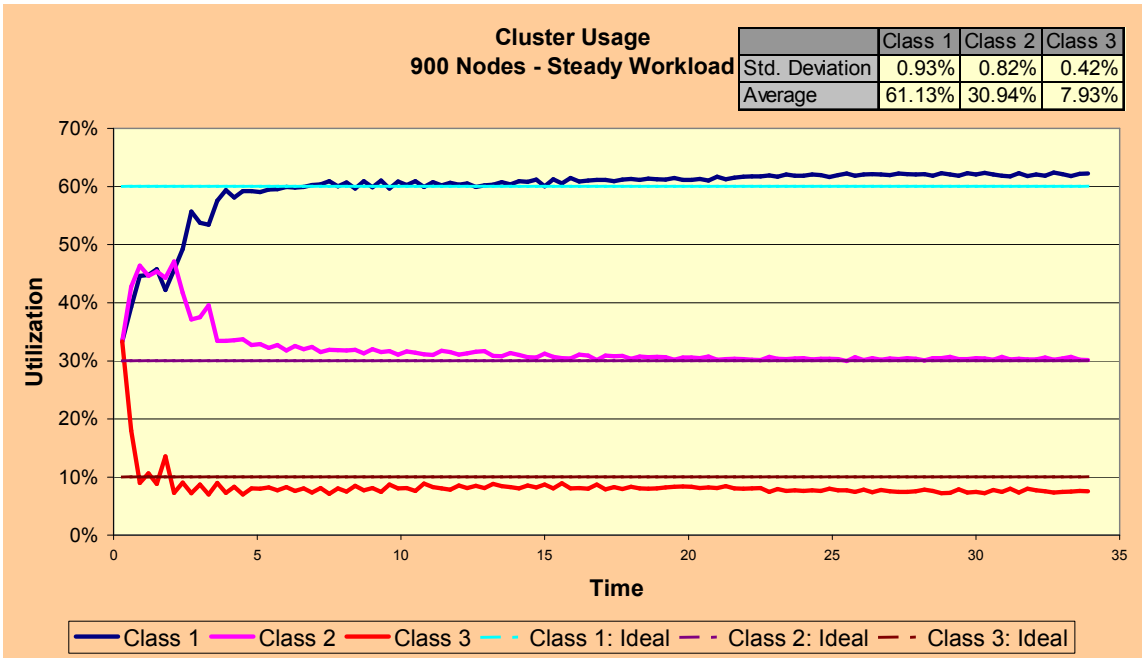

Figure 7: 900 Nodes with a steady workload

In addition, we demonstrate our management for nine hundred nodes with 30 managers at the first level and class 3 being dispatched to the first 300 nodes. This is the same basic idea as the other static tests and helps to demonstrate that our algorithm is able to handle larger scale systems with fairly good results. Figure 7 shows essentially the same graph that the other static tests have shown. The major differences are that the standard deviation has continued to decrease inversely with the cluster size at the cost of decreased averages for class 3. The additional pressure on the third class can be attributed to the algorithm leaving some allocation for possible requests on the last 600 nodes that do not receive any jobs.

The last test performed of some note is that of skewed reporting times among nodes. All of the previous results reflected nodes reporting to managers all at the same time. We simulated skewed reporting times and found that the results were the same as in our previous examples.

4.3 Dynamic Workload

The last test performed was to demonstrate our algorithm with a very dynamic workload. This test involves 100 nodes with the same 60/30/10 proportion seen before, but in this case all classes are dispatched amongst the nodes evenly. Here we twice stop all jobs for class 1 and then 1.5 seconds later restart the dispatcher, the restart represented by the vertical red lines seen in Figure 8. After dispatching resumes, our management is able to rebalance the allocation back to the desired proportions. It takes 3 reporting cycles until the usage ratios have returned to their desired allocations and this is seen as a very steep slope about 1 second after each red line in Figure 8. This disparity is caused by delay in the dispatcher starting back up along with some of the expected delay in our algorithm. We believe that this small recovery period is more than acceptable under most working conditions. Such large dips put the most pressure on our algorithm to work properly as large allocation changes can only be achieved through step 5 where we give to needy nodes, but when our previous history (allocation) indicates no usage it can take a step or two to be ready for this increase. The graph also displays a difference in the allocation between classes 2 and 3 during the two pauses. This difference can be contributed to a smaller potential workload for class 2 during the second pause, accordingly we are happy with the algorithm's adjustments in both cases. Overall, these results seem to indicate that our performance under dynamic conditions should be more than satisfactory.

Figure 8: 100 Nodes with a dynamic workload

5 Conclusions

A sizeable amount of work has contributed to resource management. Our research focused on the recently popular cluster based computing and the need for new and better techniques to give fair overall allocation to many nodes. Specifically we tried to address the bottleneck that some previous research created with single centralized managers.

Our bank algorithm demonstrated very encouraging results. We were able to achieve allocations within 2% of our desired allocations with relatively steady allocation that is highlighted by a standard deviation of only 0.4% when managing 100 nodes. The algorithm was also able to quickly adapt itself to dynamic workloads, which would be very important in production clusters. This management is able to achieve these results while running under a hierarchal scheme that helps provide scalability and fault tolerance with reasonable performance and fairness, all of these correspond to the general goals of cluster based computing.

6 Future Work

Thanks to good performance we believe that our techniques might be beneficial to the larger cluster for which we target. Unfortunately, we were only able to simulate our clusters under fairly ideal conditions that might not fairly represent the performance of real systems. To can a true understanding a real system implementation will need to be explored. Additionally, it is probably important to further explore how different design choices have affected our and other algorithms so that even better results can be achieved through fine-tuning for specific workloads and operated demands.

7 References

[1] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, **Cluster-Based Scalable Network Services,** *Proc. 1997 Symposium on Operating Systems Principles (SOSP-16)*, St-Malo, France, Oct. 1997.

[2] M. Aron, P. Druschel, and W. Zwaenepoel. **Cluster reserves: A mechanism for resource management in cluster-based network servers**. *In Proceedings of ACM SIGMETRICS 2000*, June 2000.

[3] Andrea C. Arpaci-Dusseau and David E. Culler, **Extending Proportional-Share Scheduling to a Network of Workstations**, *International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, June 1997.

[4] Andrea C. Arpaci-Dusseau, David E. Culler, Alan Mainwaring, **Scheduling with Implicit Information in Distributed Systems,** *Sigmetrics'98 Conference on the Measurement and Modeling of Computer Systems* .

[5] Banga, G., Druschel, P,. Mogul, J. **Resource Containers: A New Facility for Resource Management in Server Systems**, *Proceedings of the Third Symposium on Operating System Design and Implementation (OSDI-III)*, New Orleans, LA, February, 1999, 45-58.

[6] R. Fourer, D.M. Gay and B.W. Kernighan, AMPL: A Modeling Language for Mathematical Programming, 1993, http://www.ampl.com/.

[7] M. Mutka and M. Livny, **Scheduling Remote Processing Capacity In a Workstation-Processor Bank Network**, *Proceedings of the 7th International Conference on Distributed Computing Systems*, Sept. 1987.

[8] NS – **Network Simulator Manual**, http://www.isi.edu/nsnam/ns/ns-documentation.html.

[9] Robert Vanderbei, **LOQO User's Manual**, http://www.princeton.edu/~rvdb/.

[10] Waldspurger, C.A. and Weihl, W.E., **Lottery Scheduling: Flexible Proportional-Share Resource Management,** *Proceedings of the First Symposium on Operating Systems Design and Implementation*, Monterey CA, November 1994, pp. 1-11.