

# Resource Allocation for Autonomic Data Centers using Analytic Performance Models

Mohamed N. Bennani and Daniel A. Menascé  
Dept. of Computer Science, MS 4A5  
George Mason University  
4400 University Dr.  
Fairfax, VA 22030  
{mbennani,menasce}@cs.gmu.edu

## Abstract

*Large data centers host several application environments (AEs) that are subject to workloads whose intensity varies widely and unpredictably. Therefore, the servers of the data center may need to be dynamically redeployed among the various AEs in order to optimize some global utility function. Previous approaches to solving this problem suffer from scalability limitations and cannot easily address the fact that there may be multiple classes of workloads executing on the same AE. This paper presents a solution that addresses these limitations. This solution is based on the use of analytic queuing network models combined with combinatorial search techniques. The paper demonstrates the effectiveness of the approach through simulation experiments. Both online and batch workloads are considered.*

## 1. Introduction

Autonomic computing systems, also known as self-\* systems, can regulate and maintain themselves without human intervention. Such systems are able to adapt to changing environments (such as changes in the workload or failures) in a way that preserves given operational goals (e.g., performance goals). There has been significant research and attention to autonomic computing in the recent years [2, 3, 4, 5, 7, 8, 9, 10]. In our previous work [3, 8, 9, 10], we developed a technique to design self-organizing and self-tuning computer systems. This technique is based on the combined use of combinatorial search techniques and analytic queuing network models. In that work we defined the cost function to be optimized as a weighted average of the deviations of response time, throughput, and probability of rejection metrics relative to their Service Level Agreements

(SLAs). Others have used a control-theoretic approach to design self-configuring systems [6].

In this paper, we show how these techniques can be employed to solve resource allocation problems. More specifically, we use as an example a data center similar to the example used by Walsh et. al. in [13]. A data center hosts several Application Environments (AEs) and has a fixed number of servers that are dynamically allocated to the various AEs in a way that maximizes a certain utility function, as defined in [13]. The utility functions used in [13] depend in some cases on performance metrics (e.g., response time) for different workload intensity levels and different number of servers allocated to an AE. The authors of [13] use a table-driven approach that stores response time values obtained from experiments for different values of the workload intensity and different number of servers. Interpolation is used to obtain values not recorded in the table. As pointed out in [13], this approach has some limitations: i) it is not scalable with respect to the number of transaction classes in an application environment, ii) it is not scalable with respect to the number of AEs, and iii) it does not scale well with the number of resources and resource types. Moreover, building a table from experimental data is time consuming and has to be repeated if resources are replaced within the data center (e.g., servers are upgraded).

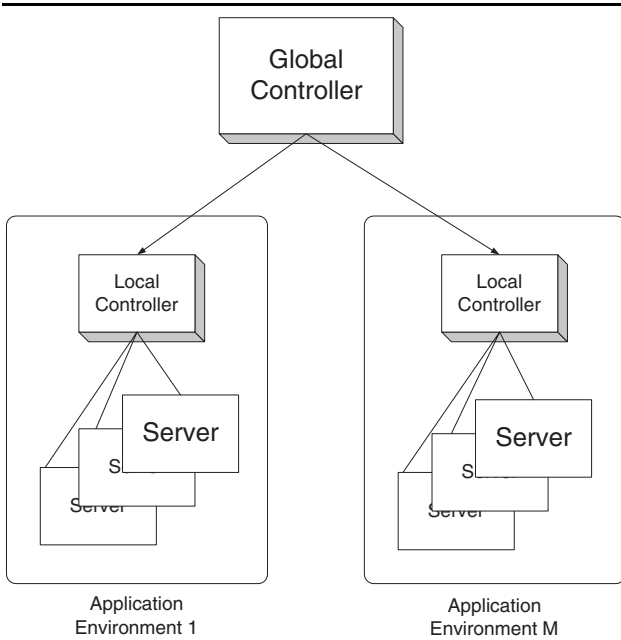
This paper proposes an alternative solution to the data center resource allocation problem along the lines of our previous work. We replace the table-driven approach with predictive multiclass queuing network models [11]. We show that the proposed solution does not suffer from the limitations mentioned above.

The rest of this paper is organized as follows. Section two formally defines the problem. Section three discusses the approach used by a resource allocation controller. This section also presents the analytic performance models used by the controller. The next section describes the experimental setting used to evaluate the approach proposed in this pa-

per. Results of the evaluation are presented in section five. Finally, section six presents some concluding remarks.

## 2. The Dynamic Resource Allocation Problem

A data center consists of  $M$  application environments (AEs) and  $N$  servers (see Fig. 1). The number of servers assigned to  $AE_i$  is  $n_i$ . So,  $\sum_{i=1}^M n_i = N$ . Each AE may execute several classes of transactions. The number of classes of transactions at  $AE_i$  is  $S_i$ . Servers can be dynamically moved among AEs with the goal of optimizing a global utility function,  $U_g$ , for the data center. A resource manager local to each AE collects local measurements, computes local utility functions, and cooperates with the global controller to implement server redeployments.



**Figure 1. Application Environments.**

AEs may run *online* transaction workloads or *batch* workloads. For the case of online workloads, the workload intensity is specified by the average arrival rate  $\lambda_{i,s}$  of transactions of class  $s$  at  $AE_i$ . For batch workloads, the workload intensity is given by the concurrency level  $c_{i,s}$  in each class  $s$  of  $AE_i$ , i.e., the average number of concurrent jobs in execution per class. We define a workload vector  $\vec{w}_i$  for  $AE_i$  as

$$\vec{w}_i = \begin{cases} (\lambda_{i,1}, \dots, \lambda_{i,S_i}) & \text{if } AE_i \text{ is online} \\ (c_{i,1}, \dots, c_{i,S_i}) & \text{if } AE_i \text{ is batch} \end{cases} \quad (1)$$

We consider in this study that the relevant performance metrics for online AEs are the response times  $R_{i,s}$  for each class  $s$  of  $AE_i$  and that the relevant metrics of batch AEs are the throughputs  $X_{i,s}$  of each class  $s$  of  $AE_i$ . We then define the response time and throughput vectors for  $AE_i$  as  $\vec{R}_i = (R_{i,1}, \dots, R_{i,S_i})$  and  $\vec{X}_i = (X_{i,1}, \dots, X_{i,S_i})$ , respectively. These performance metrics can be obtained by solving an analytic performance model,  $\mathcal{M}_i$ , for  $AE_i$ . The value of these metrics is a function of the workload  $\vec{w}_i$  and of the number of servers,  $n_i$ , allocated to  $AE_i$ . Thus,

$$\vec{R}_i = \mathcal{M}_i^o(\vec{w}_i, n_i) \quad (2)$$

$$\vec{X}_i = \mathcal{M}_i^b(\vec{w}_i, n_i) \quad (3)$$

The superscript  $o$  or  $b$  in  $\mathcal{M}_i$  is used to denote whether the model is for an online AE or a batch AE. These analytic models are discussed in Sections 3.1 and 3.2

Each AE  $i$  has a utility function  $U_i$  that is a function of the set of performance metrics for the classes of that AE. So,

$$U_i = \begin{cases} f(\vec{R}_i) = f(\mathcal{M}_i^o(\vec{w}_i, n_i)) & \text{if } AE_i \text{ is online} \\ g(\vec{X}_i) = g(\mathcal{M}_i^b(\vec{w}_i, n_i)) & \text{if } AE_i \text{ is batch.} \end{cases} \quad (4)$$

The global utility function  $U_g$  is a function of the utility functions of each AE. Thus,

$$U_g = h(U_1, \dots, U_M). \quad (5)$$

We describe now the utility functions that we consider in this paper. Clearly, any other functions could be considered. For online AEs, we want a function that indicates a decreasing utility as the response time increases. The decrease in utility should be sharper as the response time approaches a desired SLA,  $\beta_{i,s}$  for class  $s$  at  $AE_i$ . Thus, the utility function for class  $s$  at online  $AE_i$  is defined as

$$U_{i,s} = \frac{K_{i,s} \cdot e^{-R_{i,s} + \beta_{i,s}}}{1 + e^{-R_{i,s} + \beta_{i,s}}} \quad (6)$$

where  $K_{i,s}$  is a scaling factor. This function has an inflection point at  $\beta_{i,s}$  and decreases fast after the response time exceeds this value. See an example in Fig. 2 for  $K_{i,s} = 100$  and  $\beta_{i,s} = 4$ .

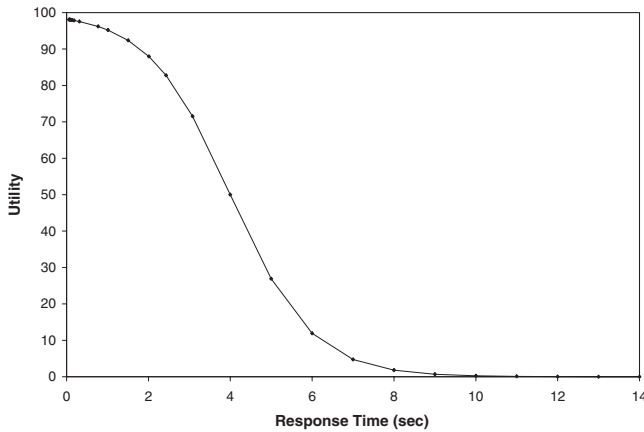
The total utility function,  $U_i$ , is a weighted sum of the class utility functions. So,

$$U_i = \sum_{s=1}^{S_i} a_{i,s} \times U_{i,s} \quad (7)$$

where  $0 < a_{i,s} < 1$  and  $\sum_{s=1}^{S_i} a_{i,s} = 1$ .

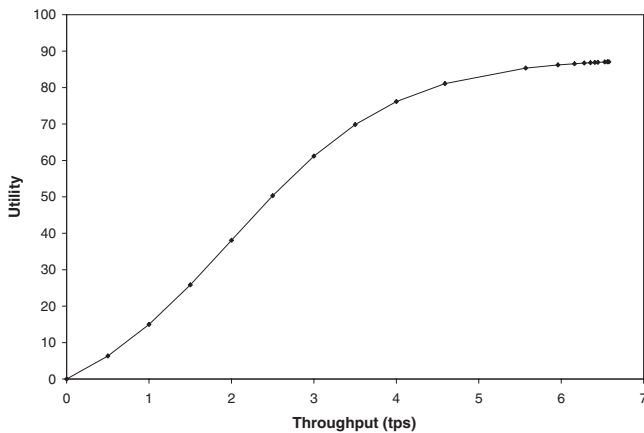
The utility function for batch AEs has to take into account the fact that the higher the throughput the higher the utility. Thus, we use the function

$$U_{i,s} = K_{i,s} \times \left( \frac{1}{1 + e^{-X_{i,s} + \beta_{i,s}}} - \frac{1}{1 + e^{\beta_{i,s}}} \right) \quad (8)$$



**Figure 2. Utility as a function of response time.**

for class  $s$  at batch  $AE_i$ . Again, as the throughput decreases from its minimum value of  $\beta_{i,r}$ , the value of the utility function goes rapidly to zero. Note that Eq. (8) is defined in such a way that the utility is zero when the throughput is zero. See an example in Fig. 3 for  $K_{i,s} = 100$  and  $\beta_{i,s} = 2$ .



**Figure 3. Utility as a function of the throughput.**

We use Eq. (7) to compute  $U_i$  in the batch case also.

The resource allocation problem addressed in this paper is similar to the one addressed in [13]. However, our solution uses a different approach, which is based on the con-

cepts we developed in our earlier work [3, 8, 9, 10]. The resource allocation problem in question is how to dynamically switch servers among AEs with the goal of maximizing the global utility function  $U_g$ .

### 3. The Controller Approach

Figure 4 depicts the main components of a local controller in an application environment. The *workload monitor* collects data about workload intensity levels (1) (e.g., transaction arrival rates per transaction class) and stores them in a workload database (2). The *workload forecaster* component uses the data in this database to make predictions about future workload intensity levels (6). Any number of statistical forecasting techniques [1] can be used here. The *predictive model solver* uses either currently observed workload measurements (5) or predicted workloads (6) along with the number of servers allocated to the AE to make performance predictions for the current workload level or for a forecast workload level. These predictions, along with SLAs for all classes of the AE, are used by the *utility function evaluator* to compute the utility function for the AE as explained in Section 2. If the utility function needs to be computed for the current configuration, measured performance metrics (response time and/or throughputs) (4) obtained from a performance monitor (3) are used instead of predictions.

Figure 5 shows the components of the global controller. A *global controller driver* determines (1) how often the *global controller algorithm* should be executed. For example, the controller could be activated at regular time intervals, called *control intervals*, or it could be made to react to changes in the global utility function. The *global controller algorithm* executes a combinatorial search technique over the space of possible configuration vectors  $\vec{n} = (n_1, \dots, n_i, \dots, n_M)$ . For each configuration examined by the controller algorithm, the corresponding number of servers (3) is sent to each AE, which return the values of the utility functions  $U_i$  (4). The *global utility function evaluator* computes  $U_g$  using Eq. 5 and returns the value (2) to the global controller algorithm. When the global controller finds a better allocation of servers, it starts the redeployment process. The breakdown between the functions of the local controller and the global controller described above is not the only possible design alternative. For example, one may want to have a lighter weight local controller and have a global controller that runs the predictive performance models for all AEs.

The global controller algorithm searches the space of configurations  $\vec{n} = (n_1, \dots, n_M)$  using a beam-search algorithm. Beam search is a combinatorial search procedure that works as follows [12]. Starting from the initial configuration, the global utility function,  $U_g$ , is computed for all

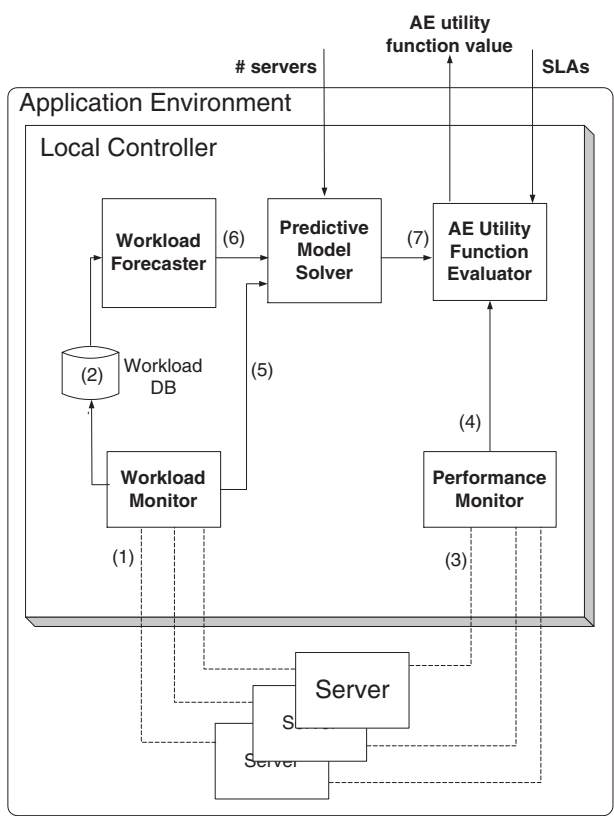


Figure 4. Local Controller.

the “neighbors” of that node. The  $k$  configurations with the highest values of  $U_g$  are kept to continue the search. The value of  $k$  is called the beam. Then, the “neighbors” of each of the  $k$  selected points are evaluated and, again, the  $k$  highest values among all these points are kept for further consideration. This process repeats itself until a given number of levels,  $d$ , is reached.

The following definitions are in order so that we can more formally define the concept of neighbor of a configuration. Let  $\vec{v} = (v_1, v_2, \dots, v_M)$  be a *neighbor* of a current configuration vector  $\vec{n} = (n_1, n_2, \dots, n_M)$ . We define  $\mathcal{N}$  as the list of possible number of servers for a batch AE in such a way that the concurrency level for all classes can be evenly distributed among all servers of that AE. This is achieved by making  $\mathcal{N}$  be the list of common denominators of the concurrency levels of all classes of that AE. For example, consider a batch AE with three classes whose concurrency levels are 30, 20, and 10. The list of common denominators, is  $\mathcal{N} = (1, 2, 5, 10)$ . Thus, if we have 5 servers, the concurrency level per server for all classes is given by (6, 4, 2). We define  $v_i$  as the successor of  $n_i$  for a batch  $AE_i$  if  $v_i$  follows  $n_i$  in  $\mathcal{N}$ . Similarly,  $v_i$  is a predecessor of  $n_i$  if  $v_i$  precedes  $n_i$  in  $\mathcal{N}$ . If  $AE_i$  is an online AE, then  $v_i$  is a suc-

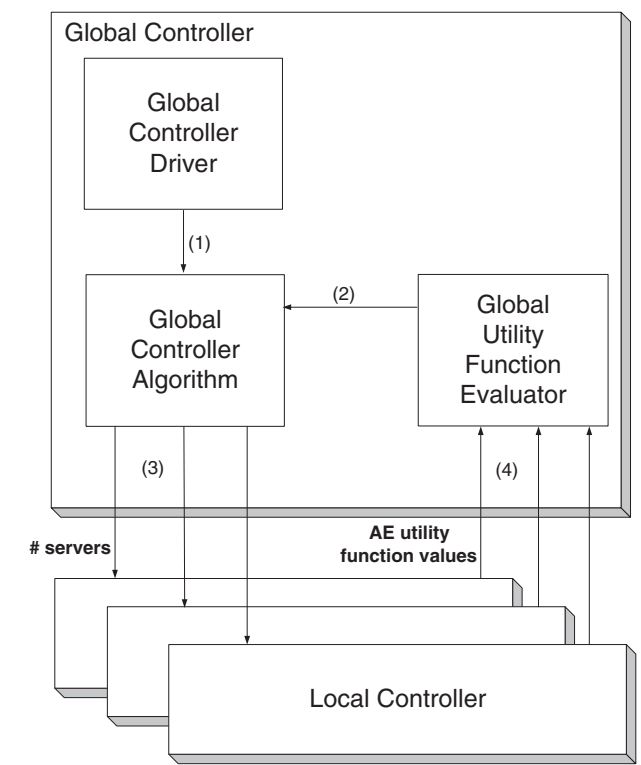


Figure 5. Global Controller.

cessor of  $n_i$  if  $v_i = n_i + 1$  and  $v_i$  is a predecessor of  $n_i$  if  $v_i = n_i - 1$ . Then,  $\vec{v}$  is a neighbor of  $\vec{n}$  iff i)  $\sum_{i=1}^M v_i = N$ , ii) the utilization of any resource within the online servers of online AEs does not exceed 100%, and iii) there is one and only one  $i (1 \leq i \leq M)$  and one  $j (1 \leq j \leq M)$  such that  $v_i$  is predecessor of  $n_i$  and  $v_j$  is a successor of  $n_j$ .

The controller algorithm is specified more precisely in Figure 6. The following notation is in order:

- $\mathcal{V}(\vec{n})$ : set of neighbors of configuration vector  $\vec{n}$ .
- $\text{LevelList}_i$ : set of configuration vectors examined at level  $i$  of the beam search tree.
- $\text{CandidateList}$ : set of all configuration vectors selected as the  $k$  best at all levels of the beam search tree.
- $\text{Top}(k, \mathcal{L})$ : set of configuration vectors with the  $k$  highest utility function values from the set  $\mathcal{L}$ .
- $\vec{n}_0$ : current configuration vector.

### 3.1. Performance Model for Online Transactions AEs

This section describes the performance model used to estimate the response time for AEs that run online transactions (e.g., e-commerce, database transactions). Let  $D_{i,s}^{\text{CPU}}$

```

LevelList0 ←  $\vec{n}_0$ ;
CandidateList ← LevelList0;
For i = 1 to d Do
  Begin
    LevelListi ← ∅;
    For each  $\vec{n} \in \text{LevelList}_{i-1}$  Do
      LevelListi ← LevelListi ∪  $\mathcal{V}(\vec{n})$ ;
    LevelListi ← Top(k, LevelListi);
    CandidateList ← CandidateList ∪ LevelListi;
  End;
 $\vec{n}_{opt}$  ← max(CandidateList)

```

**Figure 6. Controller Algorithm.**

be the CPU service demand (i.e., total CPU time not including queuing for CPU) of transactions of class  $s$  at any server of  $AE_i$  and  $D_{i,s}^{IO}$  be the IO service demand (i.e., total IO time not including queuing time for IO) of transactions of class  $s$  at any server of  $AE_i$ . In practice, the service demand at a device  $i$  for class  $s$  transactions can be measured using the Service Demand Law [11], which says that the service demand at device  $i$  is the ratio between the utilization of device  $i$  due to class  $s$  and the throughput of class  $s$ . Let  $\lambda_{i,s}$  be the average arrival rate of transactions of class  $s$  at  $AE_i$ . Then, using multiclass open queuing network models as presented in [11], the response time,  $R_{i,s}(n_i)$ , of class  $s$  transactions at  $AE_i$  can be computed as:

$$R_{i,s}(n_i) = \frac{D_{i,s}^{CPU}}{1 - \sum_{t=1}^{S_i} \frac{\lambda_{i,t}}{n_i} \times D_{i,t}^{CPU}} + \frac{D_{i,s}^{IO}}{1 - \sum_{t=1}^{S_i} \frac{\lambda_{i,t}}{n_i} \times D_{i,t}^{IO}}. \quad (9)$$

Note that the response time  $R_{i,r}$  is a function of the number of servers  $n_i$  allocated to  $AE_i$ . Eq. (9) assumes perfect load balancing among the servers of an AE. Relaxing this assumption would be straightforward.

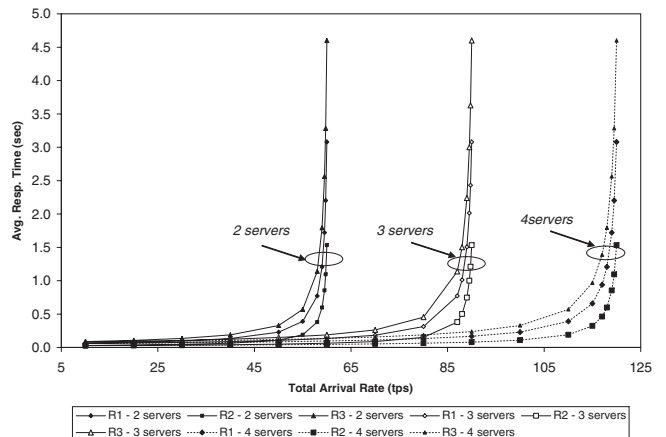
Figure 7 shows response time curves obtained with Eq. (9) for a case in which an AE has three classes and two, three, or four servers are used in the AE. The x-axis corresponds to the total arrival rate of transactions to the AE, i.e., the sum of the arrival rates for all three classes. In this example, transactions of class 1 account for 30% of all arrivals, class 2 accounts for 25%, and class 3 for the remaining 45%. The service demand values used in this case are given in Table 1.

### 3.2. Performance Model for Batch Processing AEs

Some AEs may be dedicated to processing batch jobs such as in long report generation for decision support systems or data mining applications over large databases. In

	Class		
	1	2	3
CPU	0.030	0.015	0.045
IO	0.024	0.010	0.030

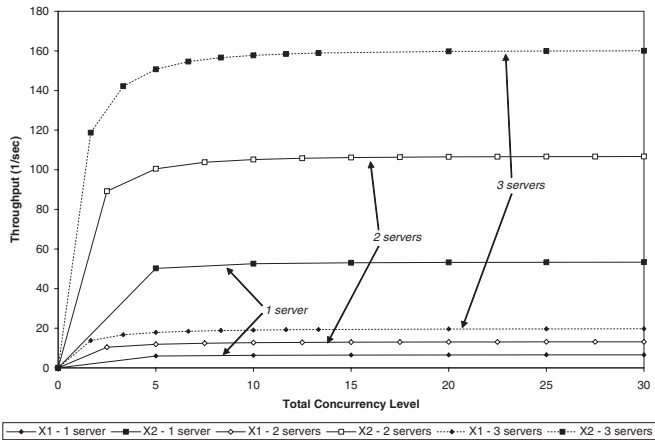
**Table 1. Service demands (in sec) for the example of Fig. 7**



**Figure 7. Response times for an Online AE.**

these cases, throughput is often of higher concern than response time. Multiclass closed queuing network models [11] can be used to compute the throughput of jobs running in batch environments. These models use multiclass Approximate Mean Value Analysis (AMVA)—an iterative solution technique for closed queuing networks. Instead of transaction arrival rates, the measure of workload intensity is the concurrency level, i.e., the number of concurrent jobs in execution in each class. We assume here that the total concurrency level of an AE is equally split among the servers of that AE.

Figure 8 shows how the throughput of two classes of batch applications running in an AE varies as the number of servers allocated to that AE varies from 1 to 3. The graphs, obtained through analytic models [11], show the variation of the throughput for each class as a function of the total concurrency level within the AE, i.e., the total number of concurrent jobs of all classes running in the AE. The two classes in this example have the same service demands as classes 1 and 2 in Table 1.



**Figure 8. Throughputs for a batch processing AE.**

#### 4. The Experimental Setting

In order to evaluate the approach described in the previous section, we set up an experimental environment that simulates a data center with three AEs and 25 servers. There are two online AEs and one batch AE. Each AE runs multiple classes of transactions/jobs. The AEs were simulated using CSIM and each simulated server has one CPU and one disk. While the data center and all its AEs were simulated in one machine, a different machine executed the controller code, which runs a beam search algorithm and uses queuing network analytic models to decide the best allocation of servers to AEs. The controller communicates with the machine that simulates the data center through Windows named pipes. A local controller in each AE collects measurements from the AE (number of allocated servers, arrival rates per class for online AEs, and concurrency levels per class for batch AEs) and sends them to the global controller. The local controller is responsible for adjusting the number of servers deployed to each AE.

Transactions for each class of the two online AEs are generated by separate workload generators with their own arrival rates. Separate workload generators for each class of each AE generate transactions for the online AEs. The batch AEs have as many threads per class as the concurrency level for that class.

In this set of experiments we considered that the switching cost is zero, i.e., servers are moved from one AE to another instantaneously. This assumption is based on the fact that all applications are installed in all servers and that the switching cost amounts to launching an application, which is in the order of a few seconds.

Application Environment 1			
Type	Online		
$S_1$	3		
$s$	1	2	3
$D_{1,s}^{CPU}$	0.030	0.015	0.045
$D_{1,s}^{IO}$	0.024	0.010	0.030
$\beta_{1,s}$	0.060	0.040	0.080
$a_{1,s}$	0.350	0.350	0.300
Application Environment 2			
Type	Online		
$S_2$	2		
$s$	1	2	
$D_{2,s}^{CPU}$	0.030	0.015	
$D_{2,s}^{IO}$	0.024	0.010	
$\beta_{2,s}$	0.100	0.050	
$a_{2,s}$	0.450	0.550	
Application Environment 3			
Type	Batch		
$S_3$	3		
$s$	1	2	3
$D_{3,s}^{CPU}$	0.005	0.010	0.015
$D_{3,s}^{IO}$	0.004	0.008	0.012
$\beta_{3,s}$	400	250	150
$a_{3,s}$	0.350	0.350	0.300
$c_{3,s}$	30	20	10

**Table 2. Input Parameters for the Experiments**

Table 2 shows the input parameters considered for each AE. The table shows the service demands in seconds for each class, the SLA for each class (the  $\beta_{i,s}$  values), and the weights ( $a_{i,s}$  values) used to compute the utility function  $U_i$  according to Eq. (7). The values of  $\beta$  for the two online AEs are given in seconds because they correspond to response times. The  $\beta$  values for the batch AE are given in jobs/sec because they correspond to minimum throughput requirements. The table also shows the values of the concurrency levels for the three classes of  $AE_3$ . These values do not change during the experiments.

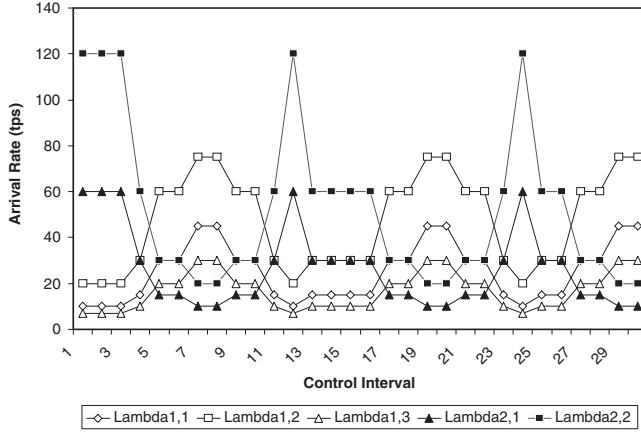
The global utility function,  $U_g$ , is computed as

$$U_g = 0.35 \times U_1 + 0.35 \times U_2 + 0.3 \times U_3. \quad (10)$$

The values of the scaling factors  $K_{i,s}$  for the online AEs were computed in such a way that the value of the utility function is 100 when the response time is zero. Thus,

$$K_{i,s} = 100 \left( \frac{1 + e^{\beta_{i,s}}}{e^{\beta_{i,s}}} \right) \quad (11)$$

for the two online AEs. For the batch AE, the scaling factor for each class was computed in a way that its value is



**Figure 9. Variation of the workload intensity for AEs 1 and 2.**

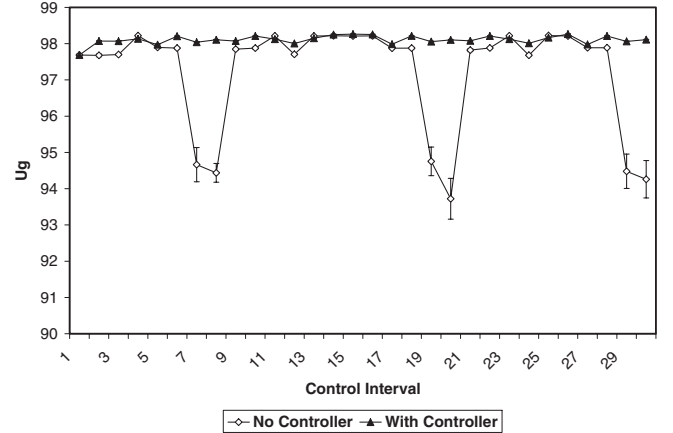
100 when the throughput for the class approaches its upper bound. For the multiple class case, we use the single class heavy-load asymptotic bound [11], which is a loose upper bound for the multiple class case. This bound,  $X_{i,s}^+$ , is equal to the inverse of the maximum service demand for that class multiplied by the maximum number of servers that could be allocated to that AE. Thus,

$$X_{i,s}^+ = \frac{N - (M - 1)}{\max(D_{i,s}^{CPU}, D_{i,s}^{IO})} \quad (12)$$

and the value of  $K_{i,s}$  for batch workloads is

$$K_{i,s} = \frac{100}{\left( \frac{1}{1 + e^{-X_{i,s}^+ + \beta_{i,s}}} - \frac{1}{1 + e^{\beta_{i,s}}} \right)}. \quad (13)$$

During the experiments, the transaction arrival rates at the two online AEs was varied as shown in Fig. 9. The x-axis is the control interval (CI), set at 2 minutes during our experiment. For AE 1, the arrival rates for classes 1–3 have three peaks, at CI = 7, CI = 19, and CI = 29. The arrival rates for classes 1 and 2 of AE 2 peak at CI = 1, CI = 12, and CI = 24. As the figure shows, the peaks and valleys for AEs 1 and 2 are out of synch. This was done on purpose to create situations in which servers needed by  $AE_1$  during peak loads could be obtained from  $AE_2$  and vice-versa. As we show in the next section, the global controller is able to switch resources from one AE to another, as needed, to maximize the global utility function.



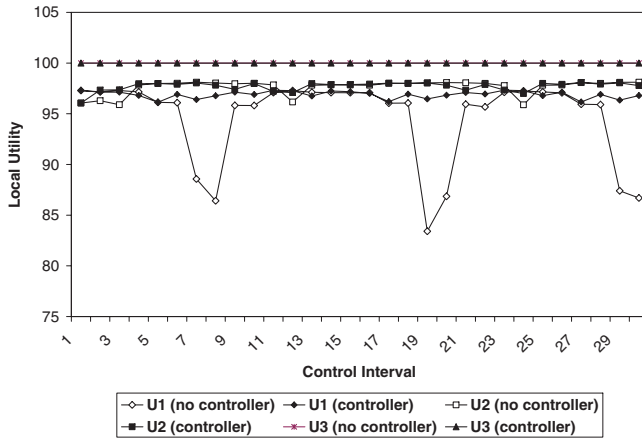
**Figure 10. Variation of the global utility function  $U_g$ .**

## 5. Results

The initial allocation of servers to AEs was set in a way that maximized the global utility function  $U_g$  for the initial values of the arrival rates. We kept the arrival rates constant at that these values and allowed the controller to obtain the best allocation of servers to AEs. The resulting initial configuration thus obtained was  $n_1 = 7$ ,  $n_2 = 8$ , and  $n_3 = 10$ .

Figure 10 shows the variation of the global utility function,  $U_g$ , during the experiment for the controller and non-controller cases. The graph also shows the 90% confidence intervals (the confidence intervals are very small for the controller case and are not as easily seen as in the non-controller case). The graph clearly indicates that the global utility function maintains its value pretty much stable when the controller is used despite the various peaks and valleys in the workload intensity for AEs 1 and 2 as shown in Fig. 9. However, when the controller is not used,  $U_g$  shows clear drops at the points where  $AE_1$  reaches its peak arrival rates and a slight reduction when  $AE_2$  reaches its peak workload intensity points.

The local utility functions are shown in Fig. 11. The figure shows that  $U_1$  and  $U_2$  decrease when the corresponding arrival rates reach their peak. The figure also shows that  $U_3$  remains flat at 100 during the whole experiment. This is due to the fact that i)  $AE_3$  is already close to its maximum possible throughput in each class and all of them exceed their SLAS ( $X_{3,1} = 548 > 400$  requests/sec,  $X_{3,2} = 363 > 250$  requests/sec, and  $X_{3,3} = 182 > 150$  requests/sec) and ii) the initialization we selected gave  $AE_3$  an adequate number of servers, which happens to be the maximum it could

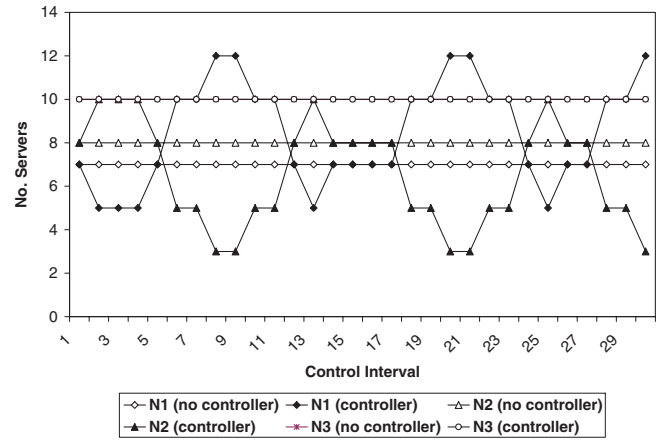


**Figure 11. Variation of the local utility functions  $U_1$ ,  $U_2$ , and  $U_3$ .**

get. Adding servers to  $AE_3$  does not improve its throughput because the utilization of the CPU (the bottleneck device for  $AE_3$ ) is already at 92%. We discuss later a situation in which  $AE_3$  participates in the movement of servers among the AEs.

Figure 12 shows the variation of the number of servers allocated to each AE during the experiment for both controlled and non-controlled cases. For the reasons explained above, the initial number of servers allocated to  $AE_3$  did not change throughout the experiment. However, AEs 1 and 2 exchange servers as needed. For example,  $n_1$  decreases from its initial value of 7 to 5 and  $n_2$  increases from its initial value of 8 to 10 at  $CI = 2$  because the controller realized that  $AE_2$ , which is at a peak, needs more servers than  $AE_1$  initially. At  $CI = 4$ , the arrival rates for  $AE_2$  start to increase and the arrival rates of  $AE_1$  start to decrease (see Fig. 9). As a consequence,  $n_1$  starts to increase and  $n_2$  starts to decrease so that at  $CI = 8$ ,  $n_1$  reaches its maximum value of 12 and  $n_2$  its minimum value of 3. Note that these values occur one CI after the respective peak and valleys for AEs 1 and 2 occur, because the controller implemented in the experiments uses the average arrival rate observed during the previous control interval as an input to the predictive performance model. The use of a forecast workload intensity might have adjusted the number of servers sooner. The pattern we just described repeats itself at  $CI = 13$ ,  $CI = 20$ , and  $CI = 30$ . As the figure shows, the controller is effective in moving servers to the AEs that need them most as a result of a varying workload intensity.

It is interesting to analyze how the response times vary as the experiment progresses and as the number of servers allocated to AEs 1 and 2 change. Figure 13 shows the variation

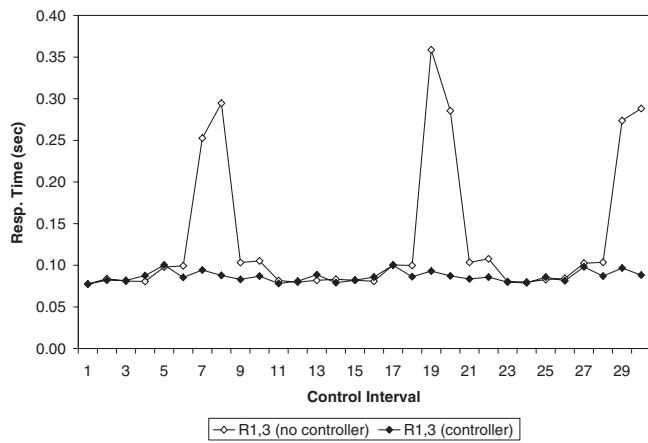
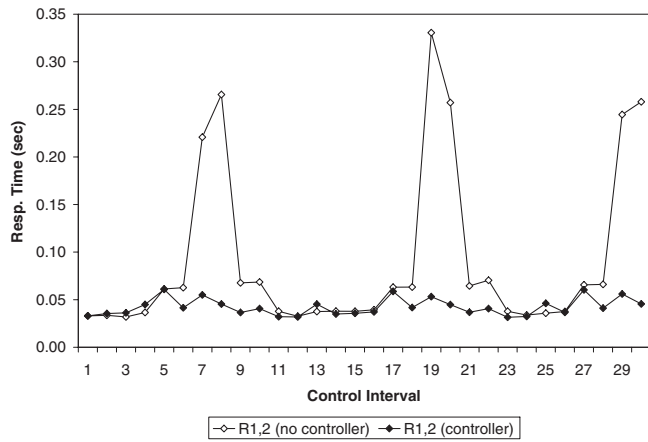
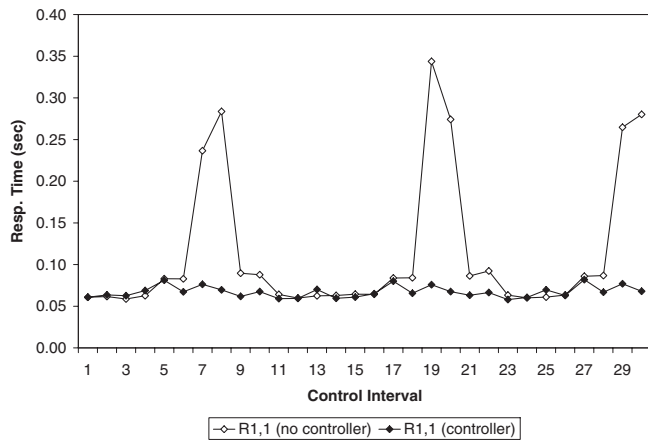


**Figure 12. Variation of the number of servers  $n_1$ ,  $n_2$ , and  $n_3$ .**

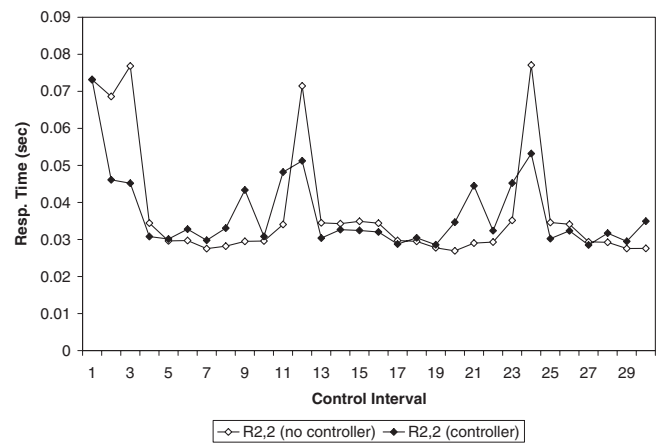
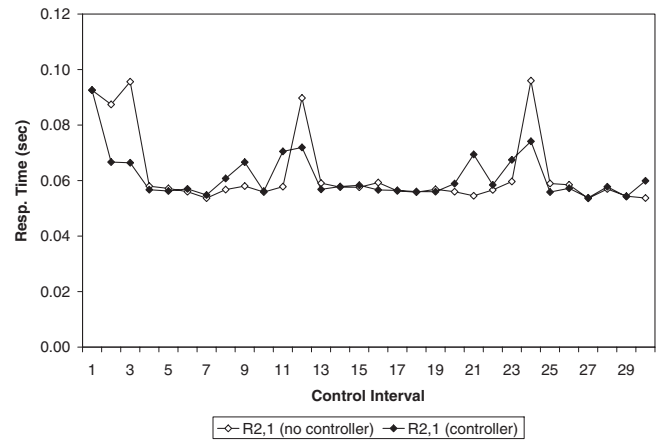
of the average response time for transactions of classes 1, 2, and 3 at  $AE_1$  for the controller and non-controller cases. As it can be seen, when the controller is used, the response times stay pretty much stable despite the peaks in workload intensity for all three classes. For example, for class 1, the response time stays around 0.076 sec when the controller is used. However, when the controller is disabled, the response time reaches three peaks at  $CI = 7$ ,  $CI = 19$ , and  $CI = 30$ , with the values of 0.284 sec, 0.344 sec, and 0.28 sec, respectively. Similar behavior can be seen for classes 2 and 3.

A similar analysis for  $AE_2$  is shown in Fig. 14 for its two classes for the controlled and non-controlled cases. As before, the non-controlled cases exhibit peaks of response times when the arrival rates of  $AE_2$  reach their peak. It should be noted also that right after  $CI = 1$ , i.e., when the controller is active, the response time for class 2 never exceeds its SLA of 0.05 sec, while the response time exceeds the SLA by 54% at the peak value without the controller.

Figure 15 shows the variation of the utilization of the CPU and disk for  $AE_1$  for the non-controller and controller case. When the controller is used, the utilization of the CPU remains in the range between 13% and 38% with the peaks in CPU utilization occurring as expected around the peaks in arrival rate. When the controller is disabled, the range in CPU utilization is much wider going from 12.6% to 56%. The same can be seen with respect to the disk. The utilization of the disk in the controlled case ranges from 9.3% to 28% while in the non-controlled case it ranges from 8.9% to 39.6%. Thus, the controller provides a much more consistent use of server resources avoiding situations of under or over utilization of resources.



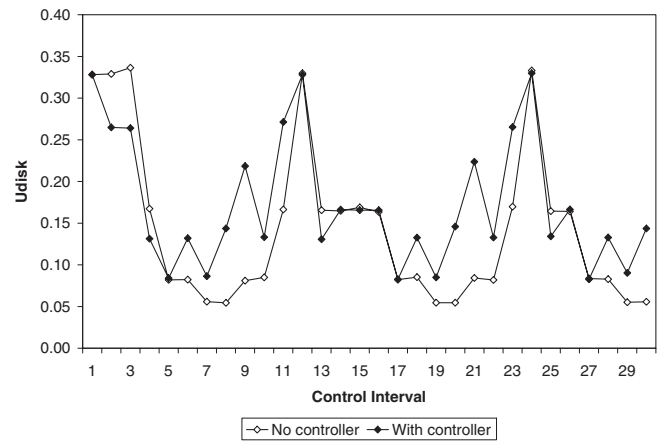
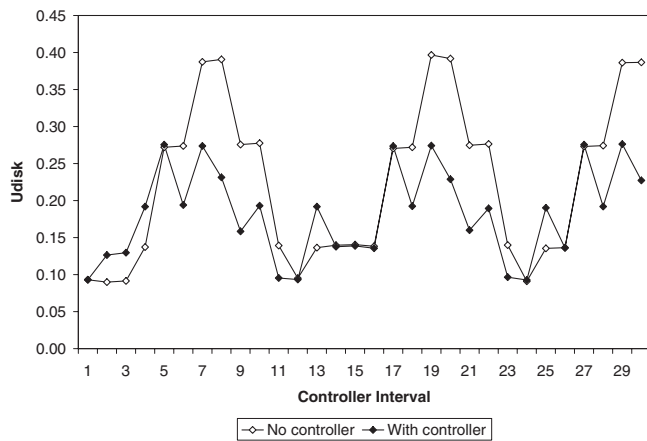
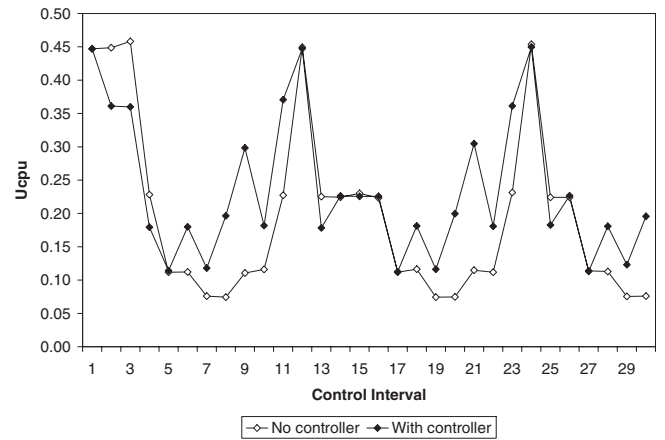
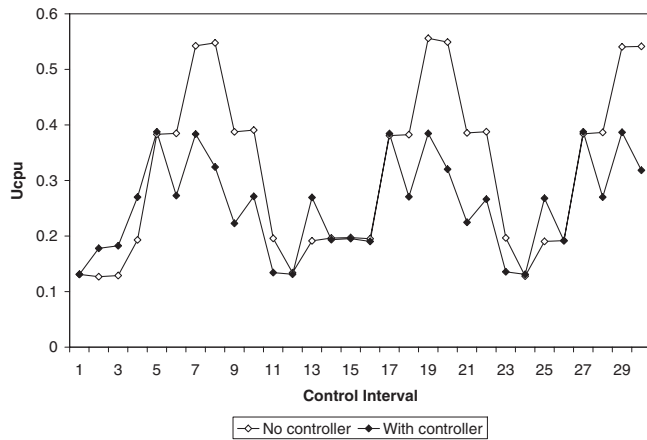
**Figure 13.** Variation of the average response times  $R_{1,1}$  (top),  $R_{1,2}$  (middle), and  $R_{1,3}$  (bottom).



**Figure 14.** Variation of the average response times  $R_{2,1}$  (top) and  $R_{2,2}$  (bottom) for  $AE_2$ .

Figure 16 shows a situation similar to that of Fig. 15, but for  $AE_2$ . Note that the utilization of the CPU and disk for  $AE_2$  is higher for the controlled case than for the uncontrolled case during most of the time. This is due to the fact that  $AE_2$  was initially allocated 8 servers, significantly more than its minimum of 3 achieved when the controller is activated.

As we discussed before, we also ran an additional experiment to force servers to be given to the batch AE,  $AE_3$ . For that purpose, we maintained the input parameters for AEs 1 and 2 and made the following changes to the parameters of  $AE_3$ :  $D_{3,1}^{CPU} = 0.004$  sec,  $D_{3,2}^{CPU} = 0.006$  sec,  $D_{3,3}^{CPU} = 0.008$  sec,  $D_{3,1}^{IO} = 0.002$  sec,  $D_{3,2}^{IO} = 0.004$  sec,  $D_{3,3}^{IO} = 0.006$  sec,  $\beta_{3,1} = 300$  jobs/sec,  $\beta_{3,2} = 200$  jobs/sec,  $\beta_{3,3} = 120$  jobs/sec,  $c_{3,1} = 18$  jobs,  $c_{3,2} = 12$  jobs, and  $c_{3,3} = 6$  jobs. These service demands are all much lower than the ones in Table 2. The new SLAs for  $AE_3$  are lower, and the concurrency levels for each class



**Figure 15. Utilization of the CPU and disk for  $AE_1$ .**

**Figure 16. Utilization of the CPU and disk for  $AE_2$ .**

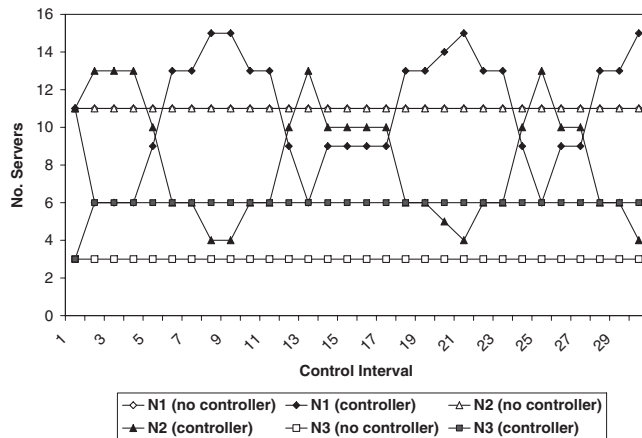
are also lower. This means that a lower initial server allocation than the one used in the previous experiment would be feasible. Thus, we gave an initial allocation of 3 servers to  $AE_3$ . Note that in this experiment, the possible number of servers for  $AE_3$  are 1, 2, 3 and 6 because of the new concurrency levels. The initial allocation is  $n_1 = n_2 = 11$ , and  $n_3 = 3$  for the same total number of servers equal to 25.

As Fig. 17 indicates, the controller gives 3 more servers to  $AE_3$  and 2 more servers to  $AE_2$  at  $CI = 2$ . These 5 servers come from  $AE_1$ . The number of servers in  $AE_3$  remains at 6, the optimal value after  $CI = 2$ , while the remaining 19 servers alternate between  $AE_1$  and  $AE_2$  with  $n_1$  ranging between 6 and 15 and  $n_2$  between 4 and 13.

## 6. Concluding Remarks

This paper addressed the problem of resource allocation in autonomic data centers. The data center hosts several ap-

plication environments that use the shared resources of the center. The resources in our case are physical servers that can run the workload of any application environment. The center has only a finite number of servers available to the application environments. Each application environment has its own SLAs. The SLAs are per class and per application environment and not per server. It is up to the data center controller to determine the required number of servers for each application environment in order to continuously meet the SLA goals under a dynamically varying workload. The degree of adherence to the SLAs is reflected by a utility function per application environment. The data center tries to maximize a global utility, which is a function of the local utility functions of the various application environments. We showed how analytic performance models can be used in an efficient manner to design controllers that dynamically switch servers from one application environment to another as needed.



**Figure 17. Variation of the number of servers  $n_1$ ,  $n_2$ , and  $n_3$  during the experiments with an initial allocation of 3 servers to  $AE_3$ .**

We would like to return to the claims we made in the introduction of the paper viz-a-viz the advantages of our method. Given that our approach is based on analytic performance models and not on simulation models, the approach scales very well with the number of AEs, resources within an AE, and transaction classes. The computational complexity of solving an open queuing network model for online AEs is  $\mathcal{O}(K S)$  where  $K$  is the number of resources (e.g, processors, disks) per server allocated to the AE and  $S$  is the number of transaction classes running on that AE [11]. The computational complexity of solving closed queuing network models for batch AEs is  $\mathcal{O}(I K S^2)$  where  $K$  and  $S$  are as before and  $I$  is the number of iterations of AMVA [11]. In most cases, AMVA tends to converge very fast (in less than 100 iterations) for tolerance values of  $10^{-5}$ . For example, an extremely large closed queuing network model with 20 resources and thirty classes and concurrency levels per class ranging from 4 to 10 was solved in 71 iterations and took 53 seconds in a Pentium 4 2.8 GHz CPU. So, if a data center has  $A_o$  online AEs and  $A_b$  batch AEs, the computational complexity of solving the models for each configuration  $\vec{n}$  is  $A_o \mathcal{O}(K S) + A_b \mathcal{O}(I K S^2)$ , which is  $\mathcal{O}(K S)$  if there are no batch AEs and it is  $\mathcal{O}(I K S^2)$  if there is at least one batch AE. At any rate, each configuration can be evaluated very fast. The number of configurations to be evaluated each time the controller runs is a function of the breadth and depth parameters used in the beam search technique.

As part of ongoing work, we are investigating the impact on the global utility function of switching servers among application environments. If switching costs are

non-negligible, the controller needs to adapt the rate at which it is invoked and the utility function needs to reflect the unavailability of a server while it is moving from one application environment to another. We are also investigating the use of adaptive controllers and workload forecasting techniques as we did in previous work.

## Acknowledgements

The work of Daniel Menascé is partially supported by grant NMA501-03-1-2022 from the US National Geospatial-Intelligence Agency.

## References

- [1] B. Abraham, J. Leodolter, and J. Ledolter, "Statistical Methods for Forecasting," John Wiley & Sons, 1983.
- [2] Babaoglu, O., Jelasity, M., Montresor, A.: Grassroots Approach to Self-Management in Large-Scale Distributed Systems. In Proc. EU-NSF Strategic Research Workshop on Unconventional Programming Paradigms, Mont Saint-Michel, France, 15-17 September (2004)
- [3] M.N. Bennani and D.A. Menascé, "Assessing the Robustness of Self-managing Computer Systems under Variable Workloads," *Proc. IEEE Intl. Conf. Autonomic Computing (ICAC'04)*, New York, NY, May 17-18, 2004.
- [4] J. Chase, M. Goldszmidt, and J. Kephart. eds., Proc. First ACM Workshop on Algorithms and Architectures for Self-Managing Systems, San Diego, CA, June 11, 2003.
- [5] J. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle, "Managing Energy and Server Resources in Hosting Centers," 18th Symp. Operating Systems Principles, Oct. 2001.
- [6] Y. Diao, N. Gandhi, J. L. Hellerstein, S. Parekh, and D. M. Tilbury, "Using MIMO Feedback Control to Enforce Policies for Interrelated Metrics With Application to the Apache Web Server," Proc. IEEE/IFIP Network Operations and Management Symp., Florence, Italy, April 15-19, 2002.
- [7] R. Doyle, J. Chase, O. Asad, W. Jin, and A. Vahdat, "Model-Based Resource Provisioning in a Web Service Utility," Fourth USENIX Symposium on Internet Technologies and Systems, March 2003.
- [8] D. A. Menascé and M.N. Bennani, "On the Use of Performance Models to Design Self-Managing Computer Systems," Proc. 2003 Computer Measurement Group Conf., Dallas, TX, Dec. 7-12, 2003.
- [9] D. A. Menascé, "Automatic QoS Control," IEEE Internet Computing, Jan./Febr. 2003, Vol. 7, No. 1.
- [10] D. A. Menascé, R. Dodge, and D. Barbará, "Preserving QoS of E-commerce Sites through Self-Tuning: A Performance Model Approach," Proc. 2001 ACM Conf. E-commerce, Tampa, FL, Oct. 14-17, 2001.
- [11] Menascé, D. A., V. A. F. Almeida, and L. W. Dowdy, *Performance by Design: Computer Capacity Planning by Example*, Prentice Hall, Upper Saddle River, NJ, 2004.

- [12] V. J. Rayward-Smith, I. H. Osman, C. R. Reeves, eds, *Modern Heuristic Search Methods*, John Wiley & Sons, Dec. 1996.
- [13] W.E. Walsh, G. Tesauro, J.O. Kephart, and R. Das, "Utility Functions in Autonomic Computing," *Proc. IEEE International Conf. Autonomic Computing (ICAC'04)*, New York, NY, May 17–18, 2004.