AUTONOMIC COMPUTING THROUGH ANALYTIC PERFORMANCE MODELS

by

Mohamed Noureddine Bennani

A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of the
the Requirements for the Degree
of
Doctor of Philosophy
Computer Science

Committee:

_____	Daniel Menascé, Dissertation Director
_____	Daniel Barbará
_____	Larry Kerschberg
_____	Robert Simon
_____	Arun Sood, Chairman, Department of Computer Science
_____	Lloyd J. Griffiths, Dean, The Volgenau School of Information Technology and Engineering
Date: _____	Spring 2006 George Mason University Fairfax, Virginia

Autonomic Computing Through Analytic Performance Models

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Mohamed Noureddine Bennani
Bachelor of Science in Computer Science
Mohammed V University, Morocco, 1994
Master of Science in Computer Science
Al Akhawayn University, Morocco, 1998

Director: Daniel Menascé, Professor
Department of Computer Science

Spring 2006
George Mason University
Fairfax, VA

Copyright © 2006 by Mohamed Noureddine Bennani
All Rights Reserved

Dedication

I dedicate this dissertation to my family, friends, and professors.

Acknowledgments

I would like to thank the following people for making this dissertation possible. First, my deep appreciation goes to my dissertation director, Dr. Daniel Menascé, for being an excellent and inspiring advisor. His commitment to excellence in teaching and research as well as his high academic and personal standards all kept me well motivated and encouraged to invest my best in my dissertation work from start to finish. Also, I would like to thank Dr. Menascé for his pleasant character, patience, tremendous availability, and readiness to help even with non research related matters. It is definitely a very enjoyable experience working with him. I certainly can never thank him enough for all he has done for me.

My profound gratitude to my doctoral committee members for the great interest they gave to my work, their constructive suggestions and recommendations, and their continuous encouragements. I am very grateful to all of them.

I am also very thankful to IBM Research for my internship with them in the summer of 2005 and for allowing me to conduct part of my research at their facility.

Many thanks to the providers of financial support for my dissertation work: the Computer Science Department, the E-Center for E-Business, and the National Geospatial-Intelligence Agency (grant number NMA501-03-1-2022).

Last but not least, I would like to thank the school's staff members, particularly Karen Alarie, Nooshi Mohebi, Anne Hamill, and Christina Leone for all their valuable assistance.

Table of Contents

List of Tables

List of Figures

Abstract

AUTONOMIC COMPUTING THROUGH ANALYTIC PERFORMANCE MODELS

Mohamed Noureddine Bennani, PhD

George Mason University, 2006

Dissertation Director: Daniel Menascé

Computing environments have gone through radical changes in the last two decades. There has been widespread production and deployment of elaborate and innovative systems and technologies. As a result, there has been a huge proliferation of new and more complex computing devices. Typically, a large number of these heterogeneous devices are interconnected to make up a large distributed system. Managing, maintaining, protecting and securing such complex systems is quite challenging even to the most skilled IT professionals. Moreover, users tend to have even stricter expectations from today's computing systems in terms of performance, availability, reliability, and security. Therefore, it is a vital necessity that current and future computer systems be built with capabilities of self-management, self-organization, self-protection, and self-healing. That is exactly the vision of Autonomic Computing. This dissertation presents a novel and robust approach to autonomic computing through analytic performance models with a greater emphasis on the self-managing and self-configuring aspects. It starts by introducing a generic architecture for a controller system that allows for self-management and self-organization and that takes into account several

design considerations for the controller including the use of workload forecasting, frequency of control, and robustness of the controller. The disseration shows how our approach to autonomic computing achieves the expected results for three instances of autonomic computing systems, namely a multi-threaded server, an Internet data center, and a virtualized server.

Chapter 1: Introduction

1.1 Motivation and Problem Statement

Computing environments have gone through radical changes in the last two decades. There has been widespread production and deployment of elaborate and innovative systems and technologies. As a result, there has been a huge proliferation of new and more complex computing devices. These devices differ from each other in a variety of ways: processing power, bandwidth requirements, component reliability, battery life, and mode of connectivity (wired or wireless). Typically, a large number of these heterogeneous devices are interconnected to make up a large distributed system. Managing such a complex system is quite challenging even to the most skilled IT professionals. Moreover, the costs of managing these complex systems increases in a non-trivial manner with the system complexity. There are several reasons for the difficulty of managing these complex systems. First, there is a significant unpredictability in current systems workloads. This is particularly the case for e-commerce systems and Web-based applications. In fact, it has been shown that, for these systems, the peak to average load ratio is very high in general [39]. Second, the architectures themselves of these systems are commonly quite complex. Usually, these architectures are multilayered or multi-tiered where the components at each layer or tier are devoted to perform different dedicated tasks. Moreover, in the context of Service Oriented Architectures (SOAs), components can be discovered at run-time. In SOAs,

components that comprise an application are designed to provide some services to other components and applications. As new services are added to or removed from the application, new components are started or stopped. Because of this dynamic change in the structure of such applications, workload characterization becomes even more difficult. Third, the emergence of new computing models such as Grid computing and Peer-to-Peer (P2P) has added to the complexity of current computer systems. In these computing models, a large number of geographically distributed heterogeneous computing elements (processors, disks, networks, etc.) make up for the resources required by massively distributed applications. Managing such computer systems is quite difficult since the number of operating computing elements may vary dynamically in relatively short periods of time.

The important technological advancements made in these last two decades in data communications and networking have contributed to the emergence of new IT based businesses. Examples include Internet Service Providers (ISPs), Data Storage Providers (DSPs), and Internet Data Centers. These businesses provide IT services to their customers usually with some guarantees on the service levels. Typically, the Service Level Agreements (SLAs) signed by both the businesses and their customers specify the quality of service (QoS) that businesses promise to provide to their customers. In general, the QoS is representative of metrics that are of utmost interest to customers. Examples of such metrics are performance-related metrics (e.g., response time and throughput), availability, reliability, and security levels. Other popular metrics focus on the business value of the services provided. Examples of such metrics include the revenue throughput and utility functions. SLAs usually stipulate the

amount of compensation that customers are entitled to in the case the hosting business cannot provide services at the levels specified by the agreement. Therefore, it is crucially important for businesses to be able to adequately manage and re-organize their computing infrastructure so that they meet the QoS levels that are desired by their customers. However, managing the QoS is often not a straightforward task as some metrics may conflict with each other (e.g., response time vs. security or response time vs. availability [47]). SLAs are more important in situations where businesses outsource the management and operation of their applications—a growing trend since it allows businesses to focus more on other aspects of their core business activities rather than on their IT infrastructure.

Another factor that makes the efficient management of computer systems such a critical issue is the tendency of reducing IT budgets in most organizations. With restricted IT budgets, organizations are expected to achieve more with even less resources. Thus, under-utilized resources need to be identified and shifted to where there is a higher computing demand. This is exactly the goal, for example, of server consolidation. With restrained IT expenditures, over-sizing the required capacity of computer systems in order to be able to offer services at acceptable levels at peak loads is not an option anymore. Finally, there is a human factor that contributes to the importance of appropriate management and re-organization of computer systems. The continuing increase in complexity of computer systems and shortage of skilled IT systems administrators makes the issue of computer system management and re-organization even more crucial. A promising way to address the obstacles to efficient management and re-organization of computer systems relies on the automation of these tasks. Therefore, it is a vital necessity that current and future computer

systems be built with capabilities for self-management and self-organization. Self-managing and self-organizing computer systems, in fact, are capable of dynamically re-configuring and re-arranging themselves in a prompt manner in response to sudden changes in the internal or external computing environments. No human supervision is needed at all. Also, because modern computer systems are made of a large number of components, the failure of some components is more likely to happen. Therefore, systems need to be able to react efficiently to failures so that they are able to maintain acceptable service levels. Systems should also be able to resume operating normally after recovering from failures. Therefore, systems should also possess a self-healing property. As far as computer system security is concerned, the increased complexity in current computer systems adds to system vulnerability to attacks. Averting security threats that can take advantage of the lack of protection of some components is increasingly difficult for a human being. Hence, future secure computer systems are also expected to have a self-protecting capability. In summary, and in the light of all the reasons presented above, computer systems need to be able to dynamically manage, re-organize, heal, and protect themselves. In other words, they should be self-managing, self-organizing, self-healing, and self-protecting. In short, they should possess the self-* properties [7].

IBM was the first to coin the term "autonomic" to describe such systems [27]. IBM's view of autonomic computing is summarized in their landmark paper [32]. In this dissertation, we also use that term. Several other major IT companies developed their own initiatives for promoting self-adaptable systems. Examples include the Adaptive Enterprise initiative by HP [24] and the Dynamic Systems initiative by Microsoft [57]. Other large IT companies are working on autonomic computing at

a smaller scale [22]. For example, Intel is working on several projects where self-adaptable systems are being developed. A representative project from Intel is the Autonomic Platform Research [29]. In its turn, Oracle, is working on an Automatic Workload Repository (AWR) for database self-management [60].

Several challenges need to be faced long before the vision of autonomic computing fully concretizes. These challenges span a wide range of technical, scientific, engineering, and architectural aspects of self-adaptable systems [33]. This dissertation constitutes an effort to face some of the challenges of autonomic systems.

1.2 Research Problem

The research problem addressed in this dissertation is that of designing and evaluating a robust and general approach to model-based controllers for self-managing computer systems and demonstrating the effectiveness of these controllers in various types of environments, namely: autonomic multithreaded servers, autonomic Internet data centers, and autonomic virtualized environments. The controllers should be able to continually determine, in a dynamic and automatic manner, the values of configuration parameters that optimize a pre-defined QoS function or a utility function for the autonomic system. Analytic performance models need to be used to guide heuristic search techniques in their exploration of the space of possible configuration values.

The next section provides an overview of the contributions of this dissertation and gives directions for future work.

1.3 Contributions

This dissertation focuses on one essential aspect of autonomic computer systems, which is self-management. More precisely, it demonstrates how analytic performance models can be beneficial for self-managing systems. The benefits are achieved through 1) an improved overall system performance and 2) a better utilization of system resources. In fact, the core concept revolves around the idea of building mechanisms into systems that allow for their dynamic reconfiguration based on the variations of the workload. To achieve this goal a computer system needs to regularly check whether it needs to re-configure itself. In this dissertation, we show how this goal can be attained through a combined use of analytic performance models that guide heuristic combinatorial search techniques in their exploration of the space of possible configurations. The performance model is in charge of evaluating and predicting, at any configuration point, the performance of the system for a given configuration point. Therefore, the dissertation starts by evaluating the impact of the particular search techniques on the performance of the system for the case of a self-managing web server and a simulated multithreaded server. In the context of a self-managing simulated multithreaded server, an assessment of the robustness of analytic performance models is conducted as well. Later, we show how the efficiency of self-managing computer systems could be further improved through the use of a combination of workload forecasting and a dynamic adjustment of the time period between consecutive triggering of the search heuristic. Another contribution relevant to the case of a self-managing multithreaded server deals with the more general case of multiple classes of customer requests. For this case, a multiclass analytic performance model is developed and

used by a self-managing multiclass multithreaded server.

This dissertation also illustrates how analytic performance models can be helpful for autonomic data centers. Here, we show how superior performance and improved resource utilization can be obtained for both the cases of a simulated data center and a realistic data center prototype. In this context, we also show how analytic performance models can efficiently cope with situations where server failures can occur.

Autonomic virtualized environments offer an additional area than can greatly benefit from analytic performance models to achieve self-management. In this dissertation, we show how an autonomic virtual server hosting multiple virtual machines can allocate the CPU intelligently to the virtual machines in a manner that maximizes its overall performance.

1.4 Organization of the Dissertation

This dissertation is organized as follows. Chapter two provides some background information and reviews basic concepts relative to queuing networks, performance metrics, workload forecasting, and heuristic search techniques. It also discusses some of the representative related work found in the literature. Chapter three presents a detailed description of our suggested mechanisms that should be built into systems to enable self-management. The chapter also provides a handy set of common performance model patterns for autonomic computing systems. Chapter four is concerned with illustrating the soundness of the proposed techniques for self-management for the case of multithreaded servers. It deals with both a simulated and a real multithreaded server. For the case, of a real multithreaded server we use a modified version

of Apache web server. In chapter five, we demonstrate the efficiency of analytic performance models for the case of autonomic data centers. Here again, we conduct the study for both the cases of a simulated data center as well as for the case of a realistic self-managing data center prototype. Chapter six demonstrates how analytic performance models can greatly improve the performance of a simulated autonomic virtual server. Chapter seven provides concluding remarks and directions for future research.

Chapter 2: Background and Related Work

This chapter starts by providing some background and definitions regarding combinatorial search techniques, forecasting techniques, and queueing networks. In particular, it reviews the definitions of the most common performance metrics such as response time, throughput, probability of rejection, and availability. The last section of the chapter discusses some work related to autonomic computing.

2.1 Background and Definitions

This section provide some background information and definitions that relate to combinatorial search techniques, queuing networks, and workload forecasting.

2.1.1 Combinatorial Search Techniques

Search is a well-known problem-solving paradigm. In fact, many real world problems are modeled and transformed to searching scenarios. The cornerstone of a search technique is to formulate the problem as that of finding a sequence of possible moves that take us from an initial state to a final one, often referred to as a goal state. This exploration of the set of all possible states, called the domain space, often takes the form of a graph traversal, mostly that of a tree traversal. Many search techniques have been studied. They are evaluated based on four criteria: completeness, optimality, time, and space complexities [75]. Completeness refers to the fact that a solution

is guaranteed to be found, if it exists. Optimality, on the other hand, reflects the quality of the returned solution, and space and time complexity measure the amount of resources that need to be mobilized to find the solution. Time and space complexity depend, most of the time, on the depth of the search tree and on its branching factor (i.e., the fanout of the tree). All search techniques fall into two broad categories. The first is the so-called uninformed or blind search techniques. These techniques do not use any outside knowledge, besides the one contained in the problem definition itself, to make decisions regarding which states (or nodes) are worth exploring first. Examples are the well-known Breadth-First Search (BFS) and Depth-First Search (DFS). The second category of search techniques is the family of heuristically informed techniques. These techniques use some extra knowledge, called heuristics, which makes finding a good solution with less effort, more likely to happen. The most famous ones are Hill-Climbing, Beam Search, and Best-First Search. A relatively newer method, called Tabu Search, belongs to this group, too. In this dissertation, we will be using hill-climbing and beam search as our heuristics of choice. Like DFS, hill climbing dives into the search tree, exploring one branch at a time. At each depth level one node is selected and pursued until no further improvements can be made or the search has reached its maximum depth limit. Hill-climbing, however, is known to suffer from some known problems, namely foothills, plains, and ridges. That is, if the domain space presents some foothills, plains, or ridges, the technique will stop progressing, move in an aimless manner, or get stuck at a local optimum, respectively.

Unlike hill-climbing, beam search expands several partial paths at the same time, as does BFS. Beam search avoids carrying out an exhaustive search by purging the remaining paths at each level. In fact, the beam factor, m, represents the number of

nodes that are explored at each level. Obviously, they ought to be the best m nodes at that depth. Other nodes, at that level, are simply ignored. Besides the beam size m, there is another parameter for the search, k, which specifies the maximum depth from the root of the tree to the nodes being explored. Hence, we refer to the technique as an (m, k) Beam Search.

2.1.2 Queueing Networks (performance metrics, availability)

In this subsection, we review the definitions of queueing networks and their most commonly used performance metrics as reported in the literature [30, 41].

Basically, a queuing network is a set of interconnected queues where each queue represents a physical resource that services customers requests. Typically, a resource is made of a server and a waiting line for customers requests. Upon the arrival of a request to a resource, it starts receiving service immediately if the server is available or it joins the waiting line otherwise. The time that a request spends receiving service at the server is referred to as the service time while the waiting time denotes the time a request spends in the line waiting to start receiving service. On the other hand, the requests inter-arrival time represents the time that elapses between consecutive arrivals of requests. Typically, a request pays several visits to the same resource before it departs from the queuing network. The total service time spent by a request at a resource over all the visits is referred to as the service demand. Usually, requests that have similar service demands are considered to be coming from the same class of customer requests. In fact, the workload imposed by a class of requests on a given system is generally characterized in terms of requests service demand and the requests arrival rate. The sum of a request's waiting time and service demand is

referred to as the request's response time. The response time is one of the main performance metrics that are of interest for a queuing networks. Other measures of interest include the throughput, the resources queue lengths, and the resources utilizations. The throughput represents the amount of requests serviced per time unit while the queue lengths represent the average number of requests in a queue at any time. The resource utilization, on the other hand, denotes the fraction of time that a resource is busy servicing customer requests. A queuing network is said to be open if there is an infinite population of customers that send requests for service, otherwise the queuing network is considered closed.

The service discipline for a queue determines the order in which the requests that arrive to the queue are serviced. Typical service disciplines include First Come First Served (FCFS), Last Come First Served (LCFS), Preemptive Priority (PP), Round Robin (RR), and Processor Sharing (PS).

Besides response time and throughput, computer systems are also evaluated by their availability, defined as the fraction of time that a system is operational. Typical values for availability are 99.9%, 99.99%, and are referred to as 3-nine availability and 4-nine availability, respectively.

2.1.3 Forecasting Techniques

Forecasting techniques are quite essential for autonomic computing. In fact, autonomic elements become more effective once empowered with forecasting abilities. The reason is that these elements are faster at noticing any changes in the trend of the system workload. They are able then to proactively position the system in a setting that would be more adequate for the coming load. Forecasting can be achieved by

applying many techniques [40]. However, no particular technique gives good forecasting results for all kind of data. There are cases when some techniques outperform others. We describe here three forecasting techniques that are very popular: weighted moving averages, exponential smoothing, and, polynomial regression [40].

- *Moving Average:* A simple and widely used forecasting method is the simple moving average (SMA). SMA makes the value to be forecast for the next period equal to the average of a number of previous observations. It is worth noticing that with this technique, only one value can be forecast at a time. A popular special case of this technique, is the weighted moving averages. Weighted moving averages is an appropriate technique for situations where the response variable (i.e., value to be forecasted) maintains an almost constant value for quite a while before changing significantly. The forecasted value is computed as a weighted average of a given number of the most recent observations. The weights are chosen in a manner that reflects the relative importance of the newest/oldest observations.

- *Exponential Smoothing:* This is another popular and simple forecasting technique known to be good for making predictions from time series data that exhibit upwards and/or downwards trends. Exponential smoothing computes a prediction as follows:

  $\text{PredictedValue} = \alpha \times \text{PreviousActualValue} + (1-\alpha) \times \text{PreviousPredictedValue}$.

  α is used to gauge the relative importance that is associated with the previous prediction as opposed to the previous observation. For example, more weight is given to observed values rather than to predicted ones if α is close to 1. A

value of α close to 0, on the other hand, gives more importance to predicted values as opposed to observed ones.

- *Polynomial Regression:* Polynomial regression is widely used as a forecasting technique due to the fact that polynomials have the nice ability of approximating fairly well any continuous function. The higher the degree of the polynomial the better is the fitting. However, in order not to introduce a severe overhead on the autonomic elements when computing the regression model, only a moderately high value for polynomial degrees should be used.

The question becomes then how to tell whether a given forecasting technique explains well the observed data. It should be noted that the overall goal is to minimize the forecasting errors (i.e., the gaps between the observed and forecasted values). Least-square methods are usually applied to properly fit the curves. As a matter of fact, a popular indicator of the goodness of the fit, called coefficient of determination and denoted by R^2, is computed based on the method of least squares [36]. R^2 takes values in the interval [0, 1]. The higher R^2 the better is the forecasting model. However, it is worth mentioning that one has to be quite careful with the least squares method. Indeed, the presence of outliers (points that do not fit the trend very well) has a significant impact on the results obtained by methods like least-squares, sometimes rendering them useless. The statistical community has recognized the need to develop robust regression methods that can generate good predictions in the presence of outliers (see for example [17, 8]). Some of these methods are already incorporated in commercial tools such as S-PLUS [65]. While the identification of real outliers is a desirable feature, the detection of "false" outliers could be very misleading. For this

reason, methods to detect unusual movements of temporal data have been proposed recently [59]. These methods are able to successfully differentiate between outliers and "plateau" changes (i.e., cases in which a new level of activity has been reached).

2.2 Related Work

The problem of designing systems that meet the performance expectations of their users and administrators has been dealt with in a variety of ways. The most representative work is the one that relates to the context of web enabled applications and e-commerce systems. For instance, the workload of information providing web sites has been extensively studied in [1, 5, 2, 12, 9, 62]. A comprehensive analysis of the workload seen by e-commerce sites is presented in [45, 43] where it was shown that these workloads tend to vary very dynamically and exhibit short-term fluctuations. Therefore, the main challenge for e-commerce sites is how to make best usage of their existing resources in order to cope with short-term fluctuations in the workload so as to maintain the QoS at its desired levels. To this end, several approaches were suggested to address this problem.

A session-admission control approach is proposed in [15]. In this technique, requests may be classified into high, medium, or low priority based on the configured policy. Priority levels are used to determine admission priority and performance-level. When the site cannot provide the desired QoS, new sessions are rejected so that the current ones can continue to experience good performance. These techniques were later incorporated into commercial products like WebQoS from HP [72]. While this approach works well for sessions in progress, it does not deal with an important QoS metric, namely the probability that a request is rejected.

Another approach to QoS control is the one incorporated in Peakstone's eAssurance product, which uses statistical models, including Bayesian and stochastic modeling to model site behavior [23, 61]. These statistical models are constantly updated based on observations of changes in applications, infrastructure, or traffic. The approach used by eAssurance suffers, however, from the following:

1. They only consider requests' response time as the sole performance metric of interest. Therefore, they do not take into account other important and potentially conflicting metrics such as throughput and the probability of rejection of requests.

2. The simple statistical models used are not robust enough to cope with the non-linear behavior of computer systems. Response time is an example of a performance metric that does not vary linearly with system workload.

Policy-based resource management is the approach adopted in [44]. The authors propose a family of resource management policies that dynamically assign priorities to customers. This approach is aimed at using the site's existing resources to optimize business metrics such as revenue throughput but does not provide guarantees in terms of QoS. So this approach does not take much into account users perceived performance.

Control theory is another approach that appears in the literature and is used for dynamic systems management. In [18], the authors present a technique for dynamically controlling the utilization of the CPU and memory of an Apache Web server. In their work, elements from control theory are used to tune the length of a session and the

maximum number of connections in order to regulate the interrelated CPU and memory utilization. This work is mainly based on the use of linear models given the fact that the metrics to control are linear. It is not known, however, how these techniques could be extended to control notoriously non-linear metrics such as response time, throughput, and the probability that requests are rejected. Moreover, no direction was given as to whether these techniques would perform well in a typical three-tier e-commerce site in which the configurable parameters at the various layers affect the site's overall QoS.

Other representative work that relate to autonomic computing in the recent years include [71, 6, 14, 13, 19, 48]. In [48], analytic performance models are used to control the QoS of an e-commerce system. It is the first work to apply queuing models for short-term systems performance optimization. Traditionally, queuing models are used in capacity planning for medium to long term provisioning of resources (in the order of few months, at least). While this work constitues a solid proof of concept, it did not deal with other control issues like the robustness of the self-managing systems, use of workload forecasting, sensitivity of the controller, or the frequency of control. We address these points in chapter 4 of this dissertation.

In [71] Walsh et. al. addressed the problem of resource allocation in an autonomic data center. In this context, 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 [71]. The utility functions used in [71] depended 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 [71] used 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 was used to obtain values not recorded in the table. As pointed out in [71], this approach has some limitations:

- It is not scalable with respect to the number of transaction classes in an application environment,
- It is not scalable with respect to the number of AEs, and
- 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).

A solution that overcomes these shortcomings is presented in chapter 5 of this dissertation.

The problem of dynamic CPU allocation in virtualized environments has been dealt with by HP through their Worklaod Manager product [26]. Workload Manager is a software solution that constitutes the goal-based policy engine in the HP Virtual Server Environment (VSE) [25]. Workload Manager ensures that the different virtual machines hosted by the VSE get adequate CPU instances (in the case of multiprocessor servers) and/or CPU shares so that either usage goals or performance goals are met. Each virtual machine specifies a range for acceptable usage or performance goals and is assigned a priority based on its relative importance. The Workload Manager regularly goes through all the hosted virtual machines in the descending order of their priorities and in incremental units of allocations either assign more units or remove

them from the virtual machine (in the case of extra capacity) until its usage or performance goals are in the desired range. However, this trial and measure approach suffers from the fact that it may take a few allocation cycles before a virtual machine finally reaches its required resource level. In chapter 6 we provide a solution to the problem of dynamic CPU allocation that does not have this shortcoming. In fact the solution we present makes usage of analytic performance models to determine, at any time, the required resouce levels for each virtual machine based on their workloads. Our technique aims at maximizing the overall utility or quality of service for the entire virtual server and not only for individual virtual machines. Moreover, our solution also allows for a dynamic change of the priorities of the virtual machines hosted by the autonomic virtual server.

Chapter 3: The General Control Approach

In this chapter we present the general control approach. Particularly, we describe the control architecture we propose for autonomic systems in the first section. We describe the components of the architecture and their interactions to reach better control decisions. QoS functions and their interpretations are presented as well. Some control decisions regarding workload forecasting, frequency of control, and distributed control are also discussed in this section. The second section of this chapter presents the most common performance patterns that are found in common autonomic systems.

3.1 Controller Approach

This section presents details of the proposed control approach. It starts by explaining the system architecture, its components, and how they interact. Then, it formally defines the QoS functions used and justifies their motivations. The end of the section also discusses some control considerations that should be taken into account when designing/deploying a control system.

3.1.1 System Architecture

Our controller approach is based on the notion that a computer system is enhanced with a QoS controller that:

1. monitors system performance,

2. monitors the resource utilization of the various resources of the system, and

3. executes, at regular intervals, called controller intervals (CI), a controller algorithm to determine the best configuration for the system (see Fig. 3.1).

As a result of running the controller algorithm, reconfiguration commands are generated to instruct the system to change its configuration.

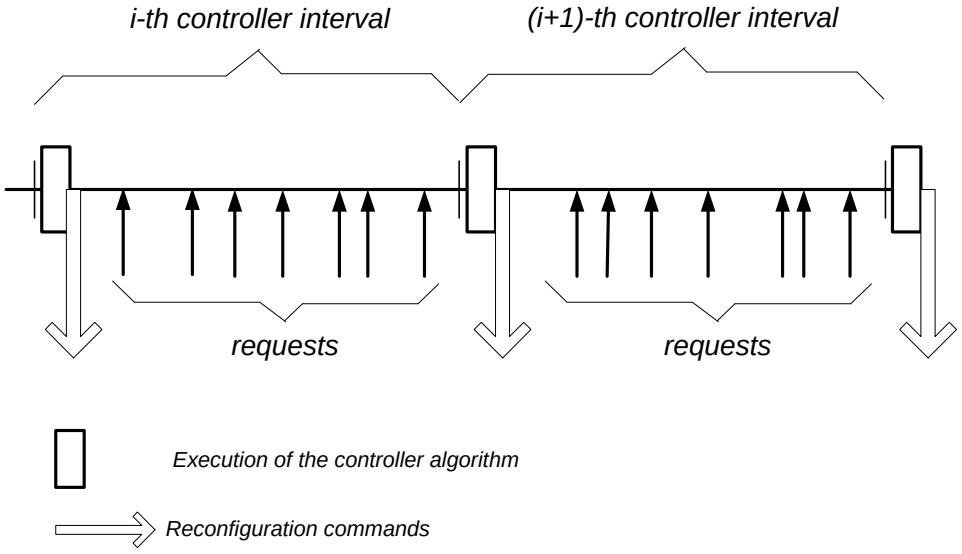


Figure 3.1: Controller intervals.

The architecture of the QoS controller is best described with the help of Fig. 3.2. The QoS controller has four main components: Service Demand Computation (2), Workload Analyzer (3), QoS Controller Algorithm (5), and Performance Model Solver (4). The Service Demand Computation (2) component collects utilization data (1) on all system resources (e.g., CPU and disks) as well as the count of completed requests (7), which allows the component to compute the throughput. The service demand of a request, i.e., the total average service time of a request at a resource, can be computed as the ratio between the resource utilization and the system throughput

according to the Service Demand Law [41]. The service demands computed by this component (8) are used as input parameters to a Queuing Network (QN) model of the computer system solved by the Performance Model Solver component (4).

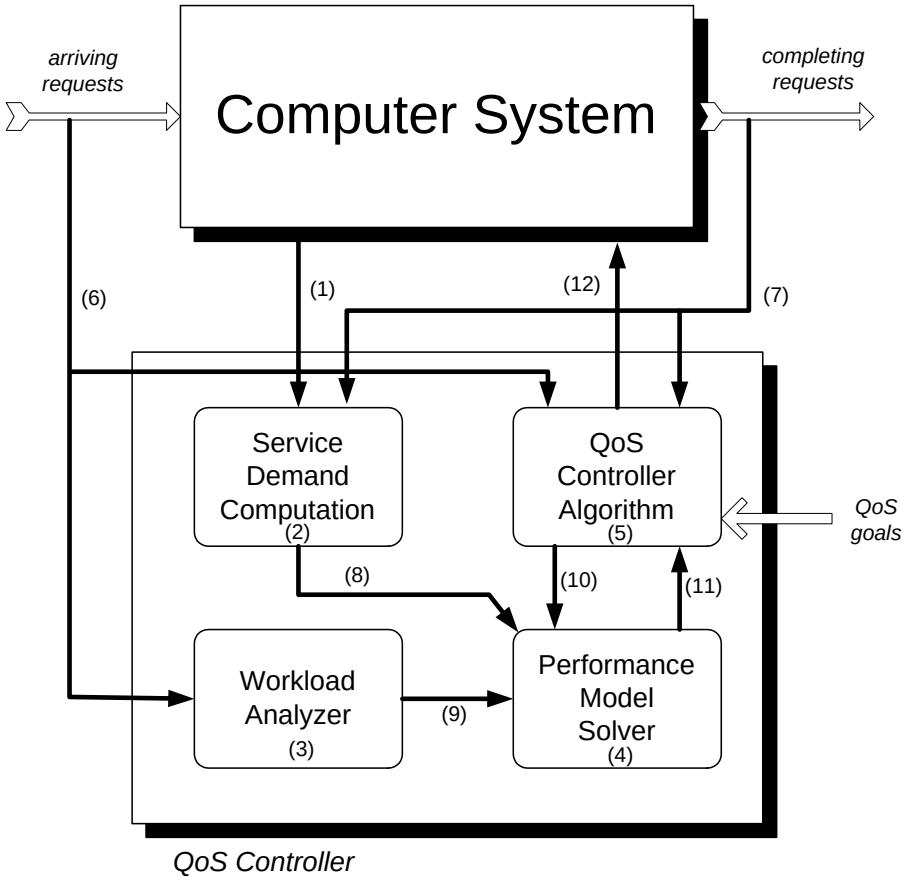


Figure 3.2: Architecture of the QoS Controller.

The Workload Analyzer (3) component analyzes the stream of arriving requests (6), computes statistics for the workload intensity, such as average arrival rate, and uses statistical techniques [3] to forecast the intensity of the workload in the next controller interval. The current or predicted workload intensity values (9) computed by this component are also used as input parameters of the Queuing Network model

solved by the Performance Model Solver component (4). This component receives requests (10) from the QoS Controller Algorithm to solve the QN model corresponding to a specific configuration of the system. This Performance Model Solver component takes as input parameters to the QN model the configuration parameter values (10), service demand values (8), and workload intensity values (9). The output of the QN model is the resulting QoS value (11) for the configuration used as input by the QoS Controller algorithm. At the beginning of each controller interval (see Fig. 3.1), the QoS Controller Algorithm (5) component runs the controller algorithm. This algorithm takes into account the desired QoS goals, the arrival and departure processes, and performs a combinatorial search (e.g., beam search or hill-climbing) [63] of the state space of possible configuration points in order to find a close-to-optimal configuration. The cost function associated to each point in the space of configuration points is the QoS value of the configuration described in section 3.1.2. This QoS value has to be computed by the Performance Model Solver for each point in the space of configuration points examined by the QoS controller algorithm. Once the QoS controller determines the best configuration for the workload intensity levels provided by the Workload Analyzer, it sends reconfiguration commands (12) to the computer system. It is worth mentioning that for some design and performance considerations, not all controller components need to reside in the same machine. In case the components are distributed across several machines, fast communication mechanisms supported by the underlying local area network (LAN) could be taken advantage of for efficient inter-component communication.

3.1.2 QoS Functions

The QoS controller attempts, at each controller interval, to optimize a QoS function or a utility function that depends on QoS metrics such as response time and throughput. This section discusses these functions in details. The QoS function considered here is a modified version of the one used in [48]. It still combines relative deviations of the average response time, average throughput, and probability of rejection, with respect to their desired goals. However, all the relative deviations are normalized in our case and as a result they only take values in the interval (-1, 1).

More specifically, the relative deviation ΔQoS_R of the average response time for configuration vector $\vec{C}$ and workload vector $\vec{W}$ is defined as

$$\Delta QoS_R(\vec{C}, \vec{W}) = \frac{R_{\max} - R(\vec{C}, \vec{W})}{\max(R_{\max}, R(\vec{C}, \vec{W}))} \tag{3.1}$$

where $R_{\max}$ is the maximum average response time tolerated and $R(\vec{C}, \vec{W})$ is the response time for configuration $\vec{C}$ and workload vector $\vec{W}$, either measured during the controller interval or predicted through the use of performance models. The definition of Eq. (3.1) has the following properties:

- $\Delta QoS_R = 0$ if the response time exactly meets its SLA (i.e., $R(\vec{C}, \vec{W}) = R_{\max}$).
- $\Delta QoS_R > 0$ if the response time exceeds its SLA (i.e., $R(\vec{C}, \vec{W}) < R_{\max}$). Given that the measured response time is at least equal to the sum $\sum_{i=1}^{K} D_i$ of the service demands D_i for all K resources (see [41]), then, using Eq. (3.1), it follows that $\Delta QoS_R \leq 1 - (\sum_{i=1}^{k} D_i)/R_{\max} < 1$.

- $\Delta QoS_R < 0$ if the response time does not meet its SLA (i.e., $R(\vec{C}, \vec{W}) > R_{\max}$). Then, from Eq. (3.1) it follows that $-1 < -(1 - R_{\max}/R(\vec{C}, \vec{W})) \leq \Delta QoS_R$

An intuitive interpretation of the definition in Eq. (3.1) is that of a relative gain (or loss) with respect to the SLA (or to the response time for a given configuration and workload). For example, if the measured response time is 3 seconds and the maximum response time is 4 seconds, then ΔQoS_R = (4-3)/4= 0.25. So, there is a gain in response time of 25% relative to its SLA. If the maximum response time is 3 seconds and the measured response time is 4 seconds, then ΔQoS_R = (3-4)/4= - 0.25. So, there is a 25% loss (i.e., a negative gain) with respect to the measured response time. In other words, it would be necessary to cut down 25% of the measured response time to meet the SLA.

The relative deviation ΔQoS_P of the probability of rejection for configuration vector $\vec{C}$ and workload vector $\vec{W}$ is defined similarly to ΔQoS_R. Namely,

$$\Delta QoS_P(\vec{C}, \vec{W}) = \frac{P_{\max} - P(\vec{C}, \vec{W})}{\max(P_{\max}, P(\vec{C}, \vec{W}))} \tag{3.2}$$

where $P_{\max}$ is the maximum probability of rejection tolerated and $P(\vec{C}, \vec{W})$ is the probability of rejection for configuration $\vec{C}$ and workload vector $\vec{W}$, either measured or predicted by a performance model. The definition of Eq. (3.2) has the following properties:

- $\Delta QoS_P = 0$ if the probability of rejection exactly meets its SLA (i.e., $P(\vec{C}, \vec{W}) = P_{\max}$).

- $0 < \Delta QoS_P \leq 1$ if the probability of rejection exceeds its SLA (i.e., $P(\vec{C}, \vec{W}) <$

$P_{\max}$).

- $-1 \leq \Delta QoS_P < 0$ if the probability of rejection does not meet its SLA (i.e., $P(\vec{C}, \vec{W}) > P_{\max}$).

The relative deviation ΔQoS_X for configuration vector $\vec{C}$ and workload vector $\vec{W}$ is defined as

$$\Delta QoS_X(\vec{C}, \vec{W}) = \frac{X(\vec{C}, \vec{W}) - X^*_{\min}}{\max(X(\vec{C}, \vec{W}), X^*_{\min})} \tag{3.3}$$

where $X^*_{\min} = \min(\lambda, X_{\min})$ is the minimum value between the average arrival rate λ and the minimum required throughput $X_{\min}$ and $X(\vec{C}, \vec{W})$ is the throughput for configuration $\vec{C}$ and workload vector $\vec{W}$, either measured or predicted by a performance model. $X^*_{\min}$ is used as the Service Level Agreement (SLA) instead of $X_{\min}$ in Eq. (3.3) because it would not make sense to expect a system to meet a given minimum throughput requirement if the workload intensity is not large enough to drive the system to that throughput level. The definition of Eq. (3.3) has the following properties:

- $\Delta QoS_X = 0$ if the throughput meets its SLA (i.e., $X(\vec{C}, \vec{W}) = X^*_{\min}$).

- $0 < \Delta QoS_P \leq 1$ if the throughput exceeds its SLA (i.e., $X(\vec{C}, \vec{W}) > X^*_{\min}$).

- $-1 \leq \Delta QoS_P < 0$ if the throughput does not meet its SLA (i.e., $X(\vec{C}, \vec{W}) < X^*_{\min}$).

We can now define a single metric as a weighted sum of the three QoS deviations defined above. Thus, $QoS(\vec{C}, \vec{W}) = w_R \times \Delta QoS_R(\vec{C}, \vec{W}) + w_X \times \Delta QoS_X(\vec{C}, \vec{W}) +$

$w_P \times \Delta QoS_P(\vec{C}, \vec{W})$, where w_R, w_X, and w_P are the relative weights, in the interval [0, 1], determined by management, to indicate the relative importance of response time, throughput, and probability of rejection. Note that the value of QoS is a dimensionless number between -1 and 1. If all three metrics meet or exceed their SLAs, $QoS \geq 0$. If $QoS < 0$, then at least one of the metrics does not meet its SLA.

3.1.3 Control Considerations

To complement the control approach, it is recommended to take some additional considerations into account when designing and deploying the controller. These considerations have a significant impact on the efficiency of the controller and on the performance of the entire system.

- Workload Forecasting:

  An adequate choice of forecasting techniques to be used by the workload predictor component would result in a higher system performance. In fact, the use of effective forecasting algorithms enables the controller to acquire a more proactive behavior. By that we mean that the controller can make better configuration decisions to accomodate the future workload. Possible techniques that could be used for workload forecasting are reviewed in Chapter 2.

- Frequency of Control:

  By enabling the controller to dynamically regulate the frequency of its invokation, the overall system performance and stability could be improved. In the case of a sudden surge in the workload, an adaptive controller algorithm is able to respond quite early to such a change in the external environment. This

results in the controller being able to position the system in a more convenient configuration before performance seriously degrades. Also, in cases when the workload becomes more stationary, there is less need to keep running the controller algorithm very frequently. This may contribute to a higher overall stability of the entire system.

- Thin/Fat Local/Global controllers:

  In the case of a quite complex autonomic system, control decisions may be made in a distributed manner. In this scheme, there will be a hierarchy of control where local (small) controllers run their own control algorithms. A global (super) controller could simply approve or disapprove the decisions made by the local controllers depending on whether some global performance criteria are met or not. The question becomes then about how much authority to grant to the local and global controllers. This is absolutely an important design issue that greatly impacts the overall system performance and therefore much attention should be devoted to this matter before an autonomic system is actually deployed.

3.2 Common Performance Patterns for Autonomic Computing Systems

In this section, we review some of the analytic performance models that could be used for several instances of autonomic systems. These analytic performance models are taken in most cases from the existing literature on the subject and are discussed here for convenience.

3.2.1 A Single Class Single Threaded Online Server with an Infinite Queue

In this system, there is an infinite population of customers that send requests to the server. All requests join the same queue at the server. The server's thread picks the request at the front of the queue, services it before it departs. The queue is infinite and therefore there is no rejection of requests. The inter-arrival and service times of the requests are typically drawn from either an exponential or a general distribution. For the case of exponentially distributed inter-arrival and service times, the performance model to use is an M/M/1 model. The solution for the M/M/1 model can be found in [41, 30]. If the inter-arrival time is exponentially distributed while the service time is arbitrarily distributed, the performance model to use is the M/G/1 model. The solution for the M/G/1 model can be found in [41, 30]. In the case of arbitrarily distributed inter-arrival and service times, the performance model to use is the G/G/1 model. The solution for the G/G/1 model can be found in [41, 30].

3.2.2 A Single Class Multi-threaded Batch Server

In this system, there is only a finite number of customers in the system at any time. All customers send requests of the same type to the server. The number of threads in the server is equal to the number of customers. Therefore, customers requests do not wait for available threads to start receiving service. As soon as a request completes its service it is immediately replaced by a new request of the same type. This fixed number of requests in the system is referred to as the server's concurrency level. Single class Mean Value Analysis (MVA) is an algorithm that is usually used for solving for the performance metrics of this system. The exact description of the

MVA algorithm can be found in [41, 30].

3.2.3 A Single Class Single Threaded Online Server with a Finite Queue

In this system, there is an infinite population of customers that send requests to the server. All requests join the same queue at the server. The server's thread picks the request at the front of the queue, services it before it departs. The queue, however, has a finite capacity, K. Therefore, requests that arrive to the server and find no room in the waiting queue are simply rejected. As a result, the probabililty of requests rejection is an additional important performance metric for this kind of systems. Based on whether the inter-arrival and service times of requests are either exponentially or arbitrarily distributed, the performance model to use for these systems can be either M/M/1/K, M/G/1/K, or G/G/1/K. The solutions for these models can be found in [41, 42].

3.2.4 A Single Class Multi-threaded Online Server with an Infinite Queue

In this system, there is an infinite population of customers that send requests to the server. All requests join the same queue at the server and are of the same type. The queue of requests is infinite and the server has a finite number of threads, J, that service the requests. Any available thread can pick the request at the front of the queue, service it before it departs. The threads compete for accessing the server's shared resouces like the CPU, disks, etc. Therefore, customer requests are subject to two kinds of contention in this type of systems. First, there is a contention

for the software resources (i.e., an available server's thread), then there is a hardware contention for the system's physical resources. The performance model for this system is based on a death-birth process in a Markov Chain where the birth rate is equal to the average arrival rate of requests, λ, and the death rate is equal to the throughput of the system. Note that the death rate becomes constant as soon as there are at least J requests in the system since the server would have reached its capacity limit. Details of the solution of this performance model can be found in [38].

3.2.5 A Single Class Multi-threaded Online Server with a Finite Queue

This system is quite similar to the model described previously (section 3.2.4) with the only exception that the server queue has a finite capacity, K, this time. Therefore requests that arrive to the server and find no room in the waiting queue are rejected. As a result, the probability of requests rejection is an important performance metric in a such system. The exact performance model solution for this system can be found in [48].

3.2.6 A Multiple Class Multi-threaded Online Server with an Infinite Queue

This system is very similar to the model described previously in section 3.2.4 with the main difference being that it supports multiple classes of customer requests. Each class of requests has its own arrival rate and service demands at the various physical resources in the system. A server's thread can service any class of requests. The performance model needs to provide performance metrics on a per class basis. It makes

use of performance models for batch systems with multiple classes of requests which are typically solved using an Approximate Mean Value Analysis (aMVA) algorithm. Details of the exact performance model solution for this system can be found in [37].

3.2.7 A Multiple Class Multi-threaded Online Server with a Finite Queue

This system is very similar to the model described previously in section 3.2.6 with the main difference being that the server's queue has a finite capacity. Therefore a request that arrives to the system and does not find room in the waiting queue is rejected. The exact performance model solution for this system is an adaptation of the algorithm found in [37] and is presented in details in Chapter 4.

3.2.8 A Multiple Class Multi-threaded Batch Server

This system is quite similar to the model described previously in section 3.2.2 with the main difference being that it supports multiple classes of requests. The concurrency level per class represents the number of requests of a class that are concurrently running in the system. Once a request finishes receiving service it is immediately replaced by a new request of the same class. The performance model to solve this system needs to provide performance metrics on a per class basis. An exact multiclass Mean Value Analysis (eMVA) can be used for solving such performance models for the case of small number of customer classes. For scalability issues, an Approximate multiclass Mean Value Analysis (aMVA) is used for providing approximate solutions for systems with large number of customer classes. Details of the eMVA and aMVA algorithms can be found in [41].

3.2.9 Other Performance Patterns Found in Common Systems Architectures

In many of today's commercial application servers, like WebShpere from IBM for example [73], the system has a dynamic pool of threads that service customer requests. Therefore, the system can either create additional threads or send some working threads back to the pool based on the observed workload. As a result, the service demands of requests in such systems are not fixed anymore. Rather, they depend on the actual load on the system. Therefore, performance models that take into account that requests service demands are load dependent should be used for such systems. Load dependent Mean Value Analysis is generally used for solving these performance models. Details of the load dependent MVA algorithm can be found in [42].

Chapter 4: Autonomic Multithreaded Servers

This chapter investigates the efficiency of the proposed control approach on a particular instance of autonomic systems, namely a multithreaded server. The choice of this system is motivated by the fact that this architecture is adopted in most of the available commercial servers in the market (web servers, application servers, database servers, etc.). The first section of this chapter considers the case of a single class of requests while the second section deals with the more general case of multiple classes of requests. In each case, analytic performance models are presented and evaluated in experimental settings and results are reported. The chapter also looks at issues like the robustness of the controller, its sensitivity to SLAs, and other controller architecture considerations such as the inclusion of a workload forecasting module and the regulation of the control frequency. Some of the work presented in this chapter appears in the following publications: [10], [46], and [47].

4.1 The Single Class Multithreaded Server Case

We describe in this section an example of a computer system that we will use to illustrate the ideas described in previous sections.

4.1.1 System Description

The computer system shown in Fig. 4.1 consists of a multithreaded server that receives requests at a rate of λ requests/sec. All requests are considered to belong to the same

class (i.e., they have similar service demands). The system has m threads and the maximum number of requests that can be in the system either waiting for a thread or using a thread is equal to n ($n \geq m$). Thus, requests that arrive and find n requests in the system are rejected. When a thread is executing a request, it is either using or waiting to use physical resources such as the CPU and disk. So, the response time of a request can be broken down into the following components: waiting for a thread (i.e., software contention), waiting for a physical resource (e.g., CPU or disk), and using a physical resource. Once a request is accepted in the system, it joins the thread waiting queue, in the event all threads are busy, and stays there until it is assigned to an available thread. Once a request completes, it leaves the system.

4.1.2 Identification of Configurable Knobs

The configurable knobs are the number of threads in the server, m, and the size of the requests queue, n. m and n are chosen because of their greatest impact on the performance of the server as also shown in [48]. In fact, it is worth noting that these two configurable parameters do not affect the performance of the system in a monotonic manner. In other words, setting the number of threads, m, to its maximum or minimum does not guarantee that system performance will be at its maximum or minimum level.

The same could be said about the requests queue size. For the case of the number of threads, m, setting it to the lowest possible value implies that requests will suffer a large delay. Also, setting m to the largest possible value might cause the requests to be serviced at a very slow rate due to the increased contention for the physical resources in addition to the possibility of a trashing behaviour of the system [41].

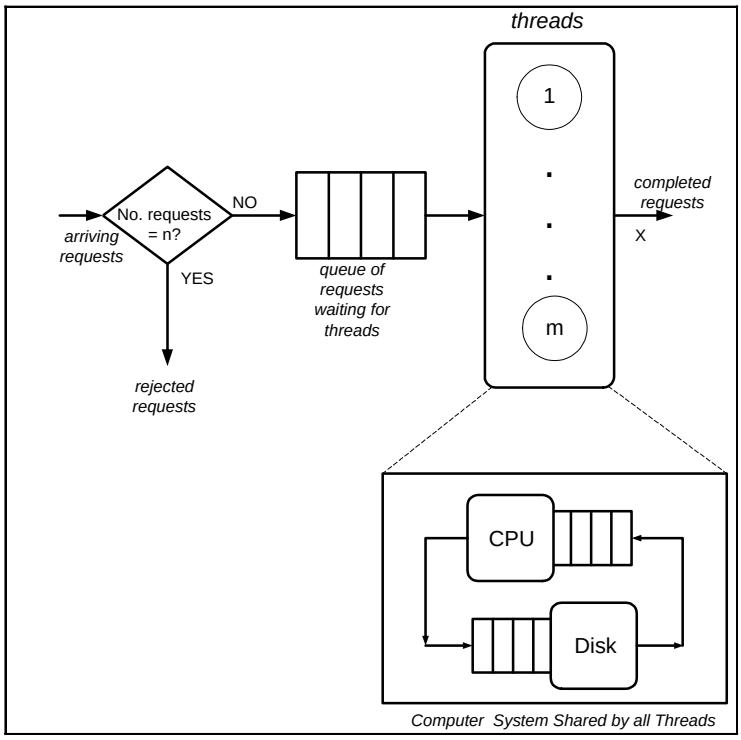


Figure 4.1: Software and hardware queues.

This phenomenon is depicted in Fig. 4.2 that also appears in [40].

Figure 4.2 shows the variation of the response time of a request as a function of m for four different arrival rate (L) values: 1 req/sec, 2 req/sec, 3 req/sec, and 3.5 req/sec. The curves of this figure were obtained by solving an analytic performance model that is explained in details in [40]. The figure shows that:

1. As the arrival rate increases, the response time increases as well.

2. There is an interplay of software contention (i.e., waiting for a software thread) and physical contention (i.e., waiting for a processor or disk). As m increases,

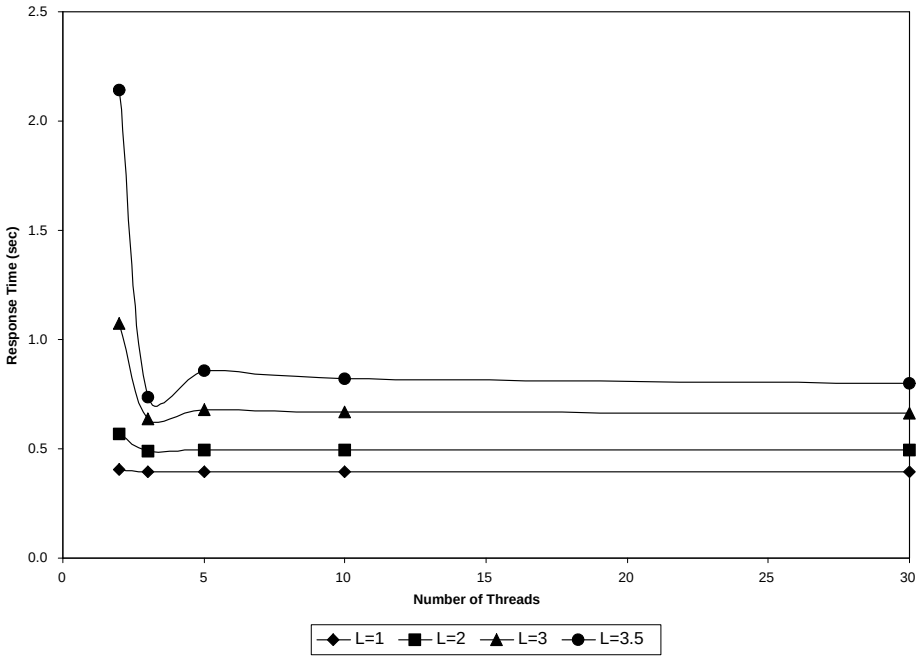


Figure 4.2: Response time vs. Number of Threads for a Multithreaded Server for Various Workload Intensity Levels (in requests/sec).

the waiting time for a software thread tends to decrease. However, at the same time, contention for physical resources increases. Therefore, depending on which factor dominates, the response time decreases or increases as a function of m. Thus, for each workload level, there is an optimal number of threads.

As for the case of requests queue size, n, setting it to the lowest possible value entails a higher probability of request rejection. On the other hand, setting n to its largest possible value could result in a very large response time due to a quite high waiting time. For these reasons, a dynamic adjustement of these two configurable parameters is the only possible option left in order to maintain system performance at acceptable levels.

4.1.3 The Single Class Performance Model

The single class analytic model used to obtain the average response time, R, average throughput, X, and probability that requests are rejected, P_{rej} , is described here. The model consists of a combination of a Markov Chain [35] and a queuing network (QN) model. The Markov Chain is used to model the waiting queue for threads and the set of m execution threads. The QN model is used to obtain the rate at which threads complete their execution. Let $X(k)$, $k = 1, ..., m$ be the rate at which a thread completes its execution when there are k requests in execution. This rate can be obtained by solving the QN model for the hardware subsystem composed of the CPU and disk when there are k concurrent threads in execution. The solution of this queuing network can be obtained through Mean Value Analysis (MVA) [40, 39, 41]. The Markov chain model of Fig. 4.3 has $n + 1$ states. A state k, for $(k = 0, ..., n)$, represents the number of requests in the system (waiting for a thread or using a thread). The arrival rate in the diagram is the arrival rate of requests, λ, and the completion rate is the completion rate of threads. Note that since there can be at most m threads in execution, the departure rate is constant and equal to $X(m)$ for any state $k > m$.

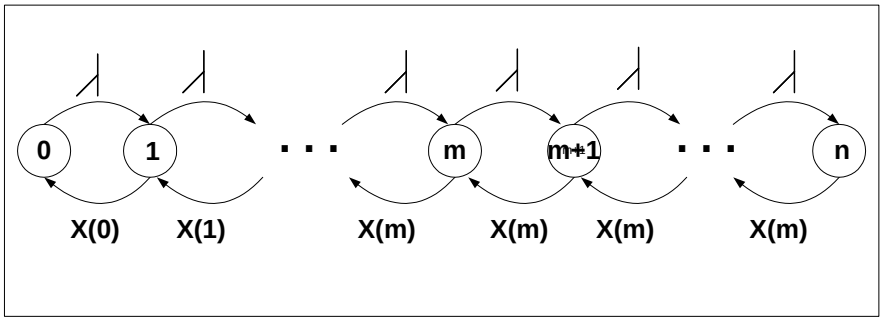


Figure 4.3: Markov Chain for the thread system.

The solution of this Markov Chain, i.e., the values of the probabilities P_k, for $k = (0, ..., n)$, of finding k requests in the system, can be obtained using the methods in [35, 40]. The solution is given below.

$$P_k = \begin{cases} P_0\lambda^k/\beta(k) & \text{for k = 1,..., m} \\ P_0\rho^k X(m)^m/\beta(m) & \text{for k = m + 1,..., n} \end{cases} \tag{4.1}$$

where:

- $\beta(k) = X(1) \times X(2) \times ... \times X(k)$,
- $\rho = \lambda/X(m)$, and
- $P_0 = [1 + \sum_{k=1}^{m} \frac{\lambda^k}{\beta(k)} + \frac{\rho \times \lambda^m \times (1-\rho^{n-m})}{\beta(m) \times (1-\rho)}]^{-1}$

The metrics of interest can be easily computed from the state probabilities and from Little's Law [35] as follows.

- $P_{rej} = P_n$
- $X = \lambda \times (1 - P_{rej}) = \sum_{k=1}^{n} X(k) \times P_k$
- $R = \frac{\sum_{k=1}^{n} k \times P_k}{X}$

4.1.4 Case of a Simulated Multithreaded Server

The simulated multithreaded server used for the experiments reported in this section has the same functional features as in the case of the real multithreaded server described in 4.1.1.

4.1.4.1 The Experimental Setting

In order to analyze the effectiveness of the controller described above, we developed a simulation program of the multithreaded server in C and C++ using the CSIM library [16]. The simulation program also implements the controller code exactly as the controller would operate in an actual system. The controller implemented in this experiment uses two different types of combinatorial search techniques: hill-climbing and beam search [**?**]. The use of hill-climbing to guide the analytic performance model in exploring the search space has been presented in [48]. However, because of the shortcomings of hill-climbing, as discussed in Chapter 2, we decided to add beam search as an additional heuristic for the controller. Our intention is to see whether these two search techniques have a significantly different impact on the performance of the controller. The two configuration parameters to be changed by the controller are the values of n and m. Thus, a configuration point is the pair (n, m) as defined previously. Hill-climbing is a very simple search technique that works as follows. Starting from the current configuration $C_0 = (n_0, m_0)$: examine all "neighbor" configurations to C_0 and move to the one with the highest value of QoS, computed by the Performance Model solver. A "neighbor" configuration is defined as one in which one of the configuration parameter values is changed by $+1$ or -1. So, the neighbor configurations of C_0 are (n_0+1, m_0), (n_0-1, m_0), (n_0, m_0-1), and (n_0, m_0+1). The search is repeated at each new point visited until either i) the value of the QoS does not improve or ii) a threshold on the number of points traversed has been exceeded. One of the drawbacks of hill-climbing is that the search can be prematurely stopped at a local optimum. An illustration of this method is shown in Fig. 4.4, which shows

various points in the space of possible configurations. A circle represents each configuration. The number within each circle indicates the QoS value of the configuration. Assume that configuration A in Fig. 4.4 is the initial configuration. The neighbors of A are B, C, D, and E. Neighbor E is the one with the highest QoS. So, the next point visited is E, which has F as the neighbor with the highest QoS value. F becomes then the next point to be visited and so on.

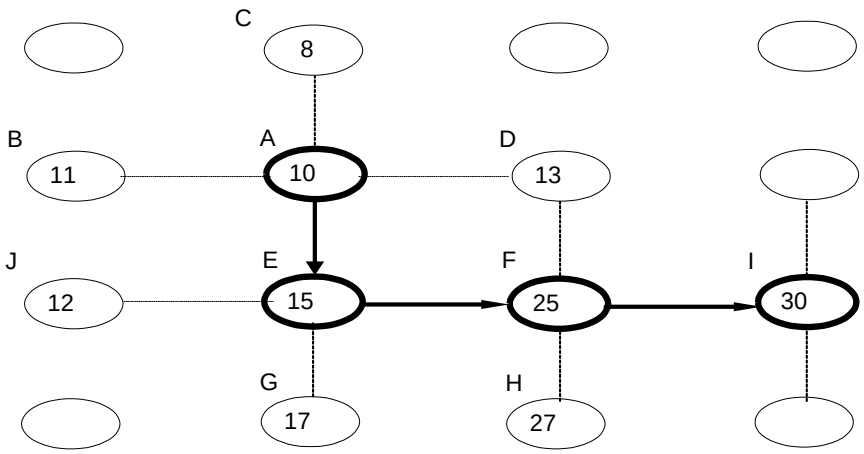


Figure 4.4: Hill Climbing.

Beam search is a combinatorial search procedure that works as follows. Starting from the initial configuration, the QoS value for all the neighbors are computed and the points with the k highest values are used 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 is reached. Then, the overall highest value is returned. Figure 4.5 illustrates how beam search

with $k = 2$ operates starting from the point at level 1. Four neighbors are evaluated but only the two with highest QoS values (15 and 18) are kept. The four neighbors of these two points are evaluated and they constitute level 2 of the tree. The two points with the highest QoS among the eight (22 and 25) are kept. Their neighbors are evaluated and configurations with QoS values 40 and 39 are selected as the two with the two highest values. In this example, the search is limited to four levels.

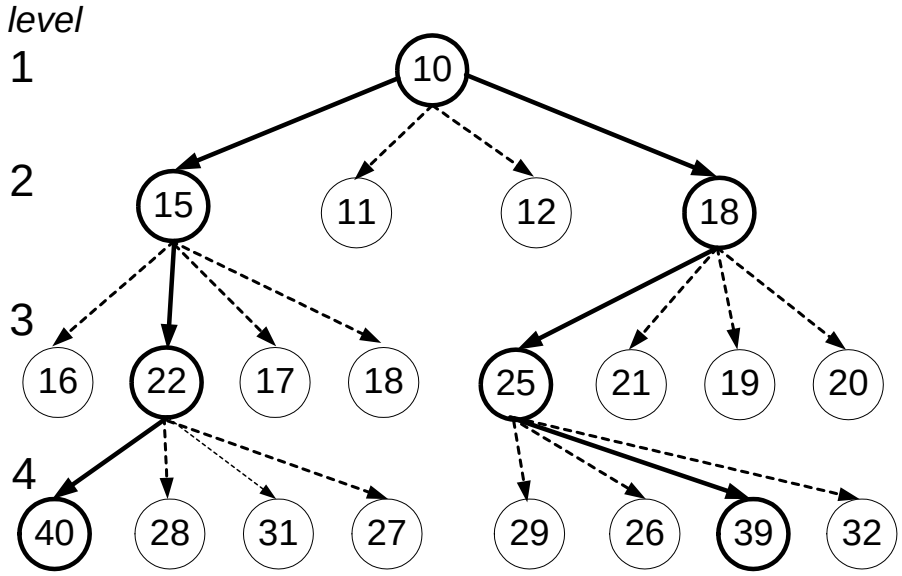


Figure 4.5: Beam Search.

Our simulation experiments consider the system of Fig. 4.1 with one CPU and one disk. The service demands at the CPU and disk are 0.03 sec and 0.05 sec, respectively. The SLAs and respective weights are:

- $R \leq 1.2$ seconds and $w_R = 0.25$,
- $X \geq 5$ requests/sec and $w_X = 0.30$,
- $P_{\text{rej}} \leq 0.05$ and $w_P = 0.45$.

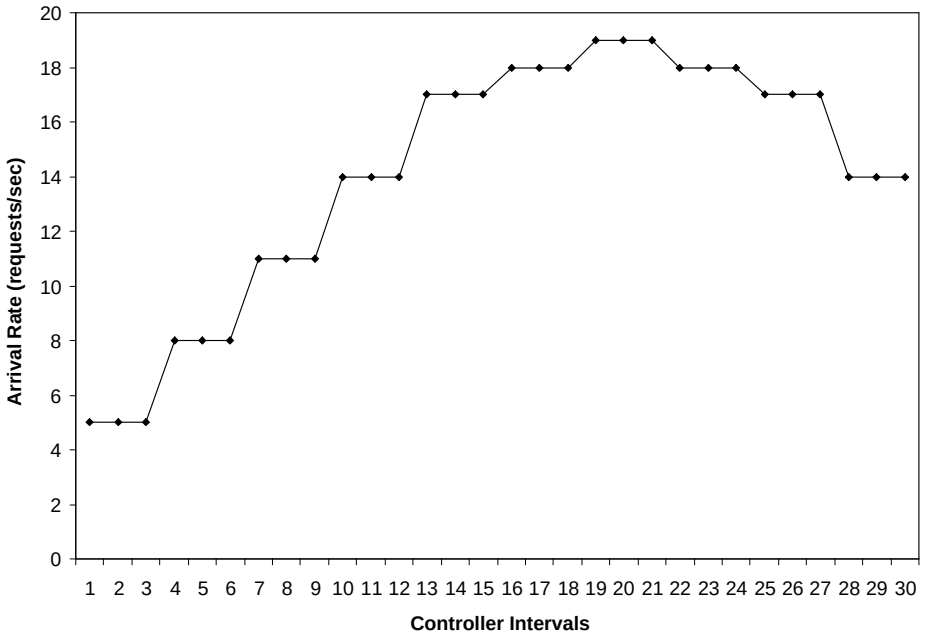


Figure 4.6: Evolution of the workload intensity.

4.1.4.2 Results

During each experiment, the arrival rate of requests started at a low value of 5 requests/sec and was increased to a peak value of 19 requests/sec and was reduced to 14 requests/sec during a period of one hour, which was divided into 30 controller intervals, as illustrated in Fig. 4.6. At the maximum value of 19 requests/sec the utilization of the bottleneck resource (i.e., the disk) is close to 100%. Therefore, we did not increase the arrival rate any further, otherwise the probability of rejection would be too high.

The controller interval duration is equal to 120 seconds. During that interval, 2280 requests arrive on average during the peak load of 19 requests/sec. Figure 4.7 illustrates the variation of the QoS metric defined in previous sections during the duration

of the experiment. The x-axis is labeled with the values of the average arrival rates observed at each controller interval. There are four curves in Figure 4.7: two of them use the two search heuristics (beam search and hill-climbing) described before, another curve corresponds to the case in which the controller is disabled, and the last one corresponds to the optimal QoS values obtained through an exhaustive search. This value is computed off-line at the end of the simulation run by examining all 9,801 possible configurations at each controller interval and determining the best one. It should be noted that for each point, the performance model has to be solved in order to compute the QoS value for that point. The two considered heuristics evaluated no more than 120 points per controller interval, i.e., 1.2% of the total number of points. In Fig. 4.7, the optimal QoS curve shows the best QoS for the next controller interval assuming that the arrival rate for that interval is the same as in the current one. The other curves show what was actually measured from the system at the beginning of each interval. These measurements are expected to be lower than the optimal values provided that the arrival rate does not change. Since the arrival rates change from interval to interval, small variations will be observed.

The following observations can be drawn from Fig. 4.7:

- At low loads, the controlled and uncontrolled cases provide almost the same value for the QoS.
- As the arrival rate increases, the controlled system manages to keep the QoS very close to its maximum value, while the QoS for the uncontrolled system falls precipitously to almost zero.
- Both beam search and hill-climbing provide almost the same value for the QoS

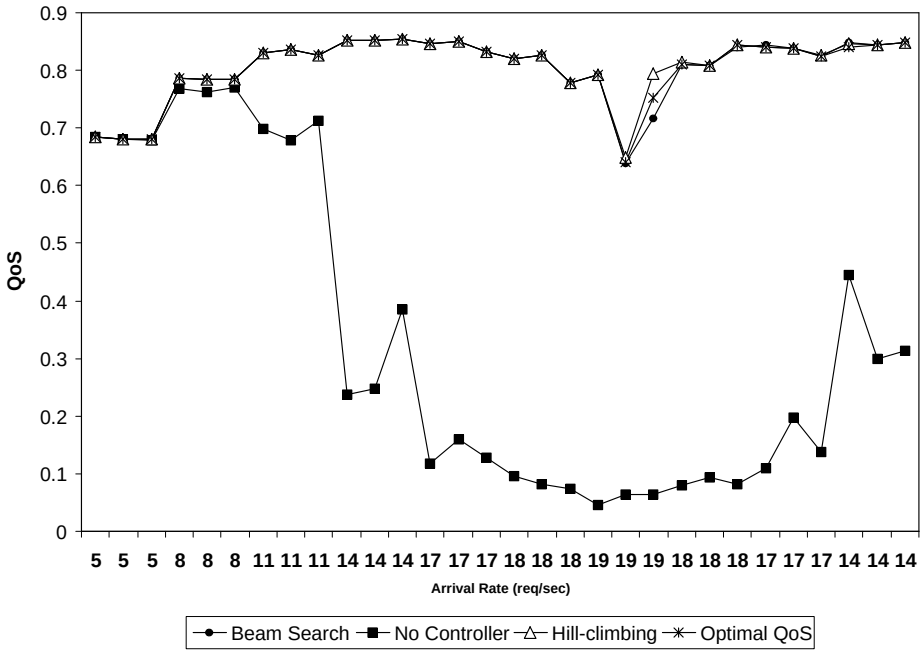


Figure 4.7: QoS values for beam search, hill-climbing, no control, and exhaustive search.

as the optimal value. In fact, the relative error between any of the two heuristics and the exhaustive search does not exceed 0.7%.

Figure 4.8 shows the utilization of the bottleneck device as a function of the arrival rate of requests. As shown in the figure, when the controller is enabled, the utilization of the disk is always higher than when the controller is turned off. In fact, as the arrival rate reaches its peak value of 19 requests/sec, the disk utilization reaches 97% for the controller case and only 77% for the uncontrolled case. The reason is that when the controller is disabled, a significant number of requests are rejected, as illustrated in Fig. 4.9. At the peak load, 21% of the incoming requests are rejected because the system is not properly configured. Figure 4.8 illustrates a very interesting feature of

the QoS controller: existing system resources can be better utilized while providing better QoS to requests.

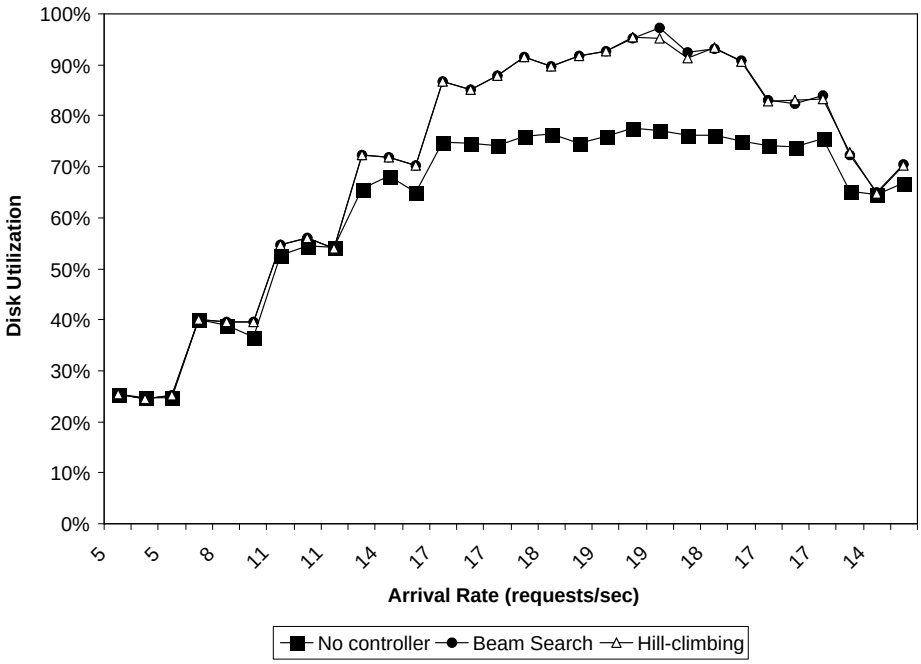


Figure 4.8: Disk utilization vs. arrival rate of requests.

Figure 4.9 shows that the controlled system adjusts itself to avoid rejecting any requests, even at high loads. The uncontrolled case violates the SLA of 0.05 for the probability of rejection as soon as the arrival rate exceeds 12 requests/sec.

Figure 4.10 shows the variation of the average response time vs. the average arrival rate of requests. As it can be seen, as the workload intensity reaches its peak, the controlled system moves towards its response time SLA of 1.2 seconds and even violates it for a very short time interval. The response time of the uncontrolled system is lower than the one for the controlled system because more than 20% of the requests are rejected at high loads and are kept out of the system. Thus, the controlled system

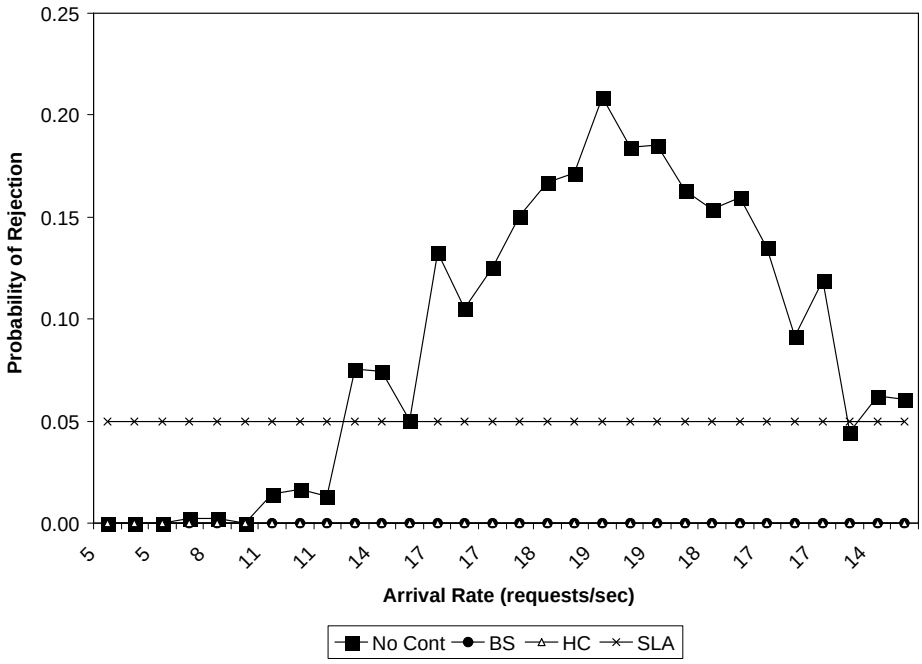


Figure 4.9: Probability of rejection vs. arrival rate of requests.

adjusts itself to meet, as close as possible, the response time SLA while minimizing the probability of rejection. This in turn provides a higher throughput as seen in Fig. 4.11.

Figure 4.11 shows that the controlled and uncontrolled systems satisfy the throughput SLA of at least 5 request/sec. However, at peak loads, the controlled system is able to process 19 requests/sec while the uncontrolled system can only process 15.5 requests/sec due to the rejected requests.

In order to complement the above experiments, we deemed it is necessary to provide an assessement of the overhead induced by the controller on the multithreaded server. Therefore, we conducted an additional set of four experiments that correspond to three different configurations for the beam search plus an experiment where an

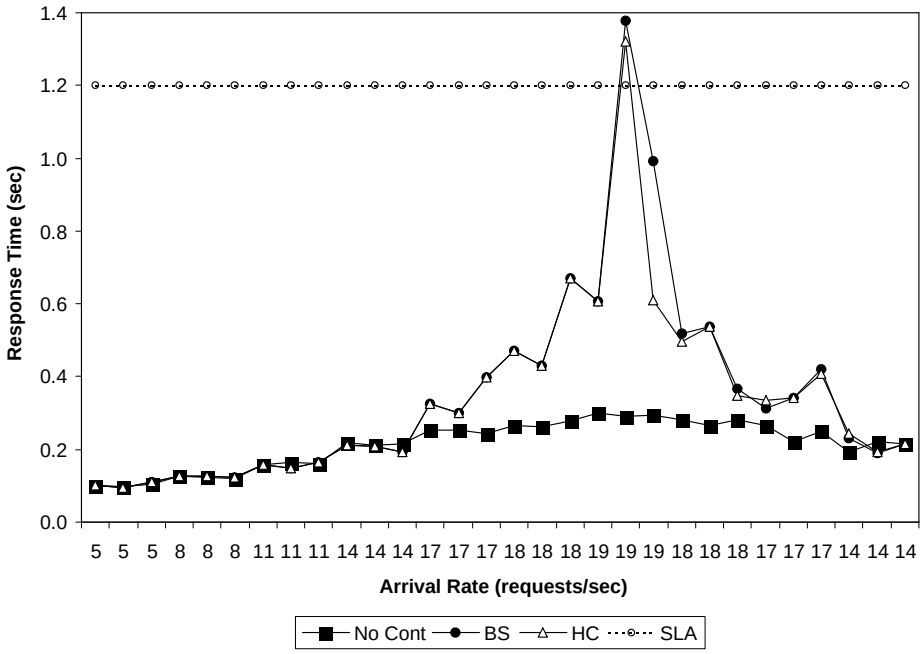


Figure 4.10: Response time vs. arrival rate of requests.

exhaustive search is used. For the three configurations of the beam search we used the same depth value ($K = 40$), but different values for the fanout; 2, 3, and 4, respectively. For each experiment, we recorded the number of points explored by the search technique as well as the time (in seconds) for the search technique to complete and return the best configuration vector to the multithreaded server.

In Table 4.1 , we report the average values obtained during the entire length of experiments (i.e., 30 controller intervals). These values were obtained for a controller running on an Intel Pentium III 800 MHz CPU. As the table shows, the average number of explored configuration vectors increases with respect to the fanout and so does the average time for the completion of the search technique. Even for a value of the fanout, m, as high as 4, we see that the average number of configuration

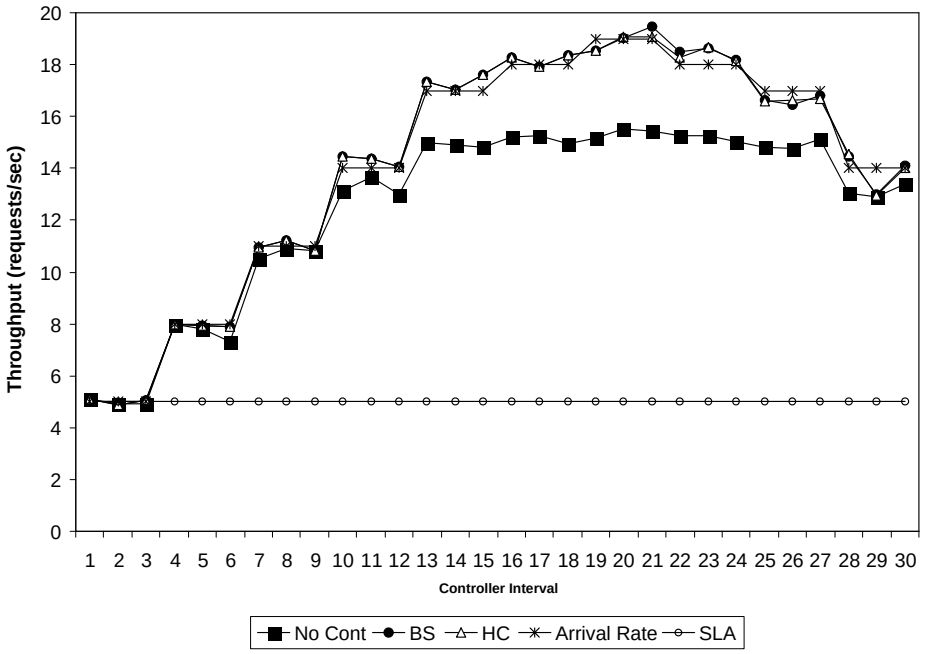


Figure 4.11: Throughput and arrival rate vs. controller interval.

vectors that are explored by the beam search is less than 2.5% of the total number of possible configuration vectors. The average time taken by a beam search with a fanout $m = 4$ is as well less than 2.5% of the average time required for an exhaustive search. Therefore, we conclude that a controller that uses an efficient heuristic search technique, like beam search for instance, is able to result into a very performant controlled system yet only with a very minimal overhead.

4.1.5 Case of a Real Web Server

4.1.5.1 The Experimental Setting

In this section we show the results of applying the techniques described above to the QoS control of an actual Web server. The HTTP server is Apache 1.3.12, which was

Table 4.1: Quantification of the Controller Overhead

Controller Overhead				
Beam Search				Exhaustive Search
K	40			
m	2	3	4	
Visited Points	144	188	228	9801
Search Time (sec)	0.025	0.038	0.049	2.15

modified to allow for a dynamic change of the number of active threads (m) and the maximum number of requests in the system (n). These parameters, m and n, are the controlled parameters. The workload used to drive the server is generated by SURGE, a workload generator for Web servers [9], using two client machines sending requests to a third machine that runs the Web server. A fourth machine runs the QoS controller. All four machines are Intel-based and run either Windows 2000 Professional or Windows XP Professional. All machines are connected through a 100-Mbps LAN switch.

SURGE was selected as the workload generator because it was demonstrated in [9] that, unlike other Web server benchmarks, it exercises servers in a manner that is consistent with actual empirical distributions observed in Web traffic. In fact, SURGE is capable of generating references in a manner that matches empirical measurements regarding file size distributions, relative file popularity, embedded file references, and temporal locality of references.

In this experiment again, the primary responses are the response time of an HTTP request (R), the throughput of the HTTP server (X_0), and the probability that a request is rejected (P_{rej}). The goal function is the same QoS value defined in the previous section. The SLAs and respective weights for the experiment described here

are: $R \leq 0.3$ seconds, $w_R = 0.5$, $X_0 \geq 50$ requests/sec, $w_X = 0.2$, $P_{\text{rej}} \leq 0.05$, and $w_P = 0.3$.

4.1.5.2 Results

Figure 4.12 shows the variation of the QoS during the experiment. The x-axis is a time axis labeled in units of control intervals. The workload intensity started at 5 requests/sec and climbed to 19 requests/sec at CI = 19. Then, the workload intensity was reduced to 14 requests/sec. The experiment in question lasted 30 CIs and each CI is equal to two minutes. Results are shown for two types of combinatorial search techniques: hill climbing and beam search (see top two curves). As it can be seen, the QoS for the uncontrolled Web server (bottom curve) becomes negative when the load reaches its peak value, indicating that at least one of the three metrics is not meeting its SLA. On the other hand, the QoS for the controlled Web server always remains in positive territory for both hill-climbing and beam search. We noticed in the various experiments we carried out that beam search tends to provide slightly better results than hill-climbing. This is probably due to the fact that the latter combinatorial search technique may at times be trapped at local optima. However, the difference between the two techniques was never significant.

4.1.6 Robustness of the Controller

Many real workloads exhibit some sort of high variability in their intensity and/or service demands at the different resources. Therefore, it is very important to investigate the behavior of the proposed technique for self-managing computer systems in such environments. The goal is to provide an assessment of the efficiency of the

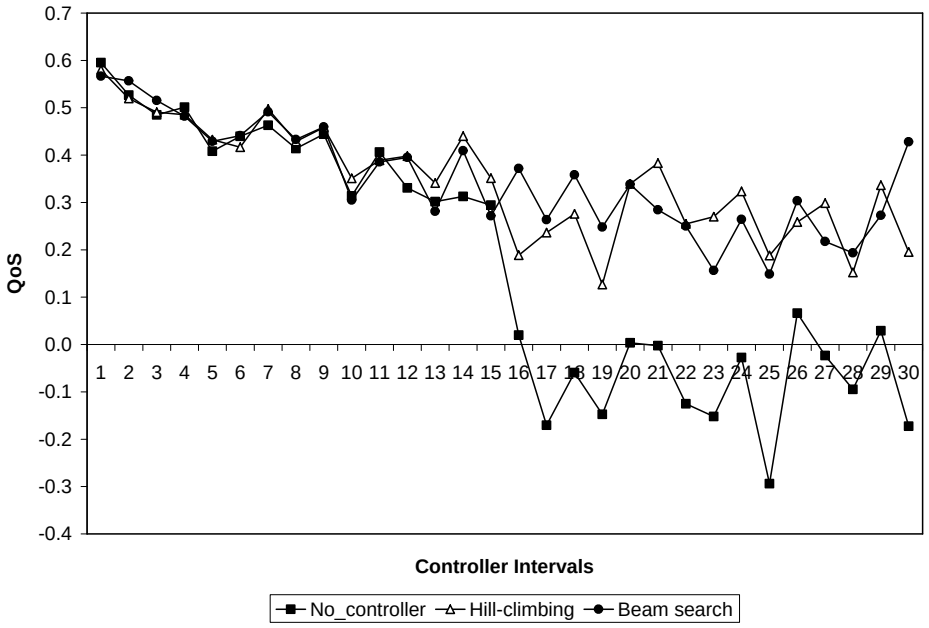


Figure 4.12: A controlled Web server

controller when the workload deviates from the underlying assumption made by the performance model of an exponentially distributed inter-arrival time and service time for the requests. To this end, we conducted a set of experiments to study the impact of the variability in the request inter-arrival and service times distributions at both system resources (i.e., cpu and disk). The variability of these distributions is represented by their respective coefficients of variation (COV) (i.e., the standard deviation divided by the mean): C_a and C_s. We used the values 0.5, 1.0, 2.0, and 4.0 for C_a and C_s for a total of 16 combinations of the values of these two coefficients of variation. We do not consider values for C_a and C_s larger than 4.0 as we are interested only in values that represent a deviation from $C_a = C_s = 1$. Moreover, a value for the squared coefficient of variation (COV^2) as large as 10 is typically considered to provide a moderately high enough variability [20].

4.1.6.1 Generating Distributions with Varying Coefficients of Variation

To generate distributions with the desired values for the coefficient of variation, we used phase-type distributions. For instance, to synthesize a distribution with a given mean μ and a COV = 0.5, we used a 4-stage Erlang distribution, where each stage has an exponentially distributed time with an average equal to $\mu/4$ [35]. For a distribution with a COV = 1, we used the exponential distribution (the only distribution that has a COV = 1). For values of the COV greater than 1, we used a 2-stage Coxian distribution [35], where each stage is exponentially distributed with an average equal to μ_i for i = 1, 2 (see Fig. 4.13). As shown in the figure, one moves from stage 1 to 2 with probability (1 - q). One can exit the server right from stage 1 with probability q ($0 < q < 1$). The average time spent in this server is equal to

$$\mu = \mu_1 + (1 - q)\mu_2. \tag{4.2}$$

The variance is given by

$$\sigma^2 = \mu_1^2 + (1 - q^2)\mu_2^2. \tag{4.3}$$

Therefore, the COV is given by:

$$COV = \frac{\sigma}{\mu} = \frac{\sqrt{\mu_1^2 + (1 - q^2)\mu_2^2}}{\mu_1 + (1 - q)\mu_2}. \tag{4.4}$$

The question becomes, then, how to choose μ_1, μ_2, and q to obtain a distribution with given values μ and COV for the mean and the coefficient of variation. First, we start by using Eq. (4.2) to write μ_1 in terms of μ_2 and q as follows: $\mu_1 = \mu - (1 - q) \times \mu_2$. Second, by replacing this expression for μ_1 in Eq. (4.4) one obtains the following quadratic equation on the unknown μ_2:

$$2(1 - q)\mu_2^2 - 2\mu(1 - q)\mu_2 + (1 - COV^2)\mu^2 = 0. \tag{4.5}$$

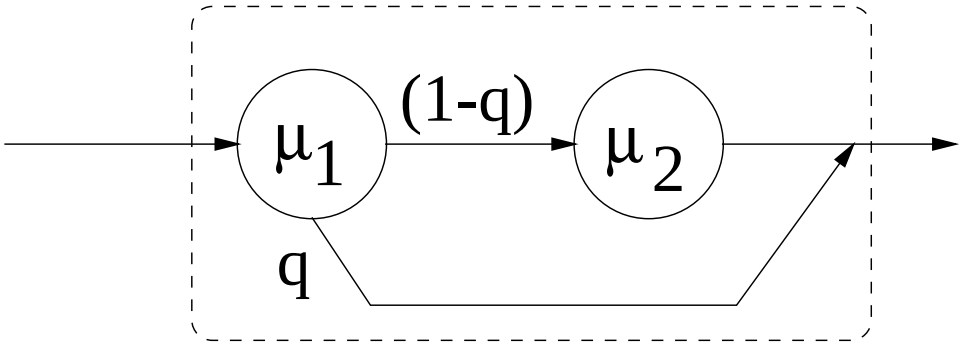


Figure 4.13: Two-phase Coxian distribution.

We can now solve Eq. (4.5) for q varying from 0.1 to 0.95 in increments of 0.05 and choose one of the values of q that results in a positive value for μ_1 and μ_2.

4.1.6.2 Experimental Setting

The experimental setting used in this section is very similar to that used in the simulated multhithreaded server section. The only difference worth mentioning is that, now, the inter-arrival time and service time distributions have coefficients of variation, C_a and C_s, respectively. The values used for C_a and C_s are: 0.5, 1, 2, and 4.

The SLA values and their corresponding weights are:

- $R_{\max} = 1.2$ seconds, $w_R = 0.35$,
- $X_{\min} = 5$ requests/sec, $w_X = 0.25$, and
- $P_{\max} = 0.05$, and $w_P = 0.4$.

The initial values for n and m are $n = 7$ and $m = 2$.

4.1.6.3 Results

Figure 4.14 depicts the variation of the workload intensity, measured in requests/sec, as a function of time, measured in controller interval units for the experiments related to the variability of the inter-arrival and service time of requests. The duration of each experiment was 30 CIs (i.e., 60 minutes since each CI was set to 2 minutes). The mean service demands at the cpu and the disk were 0.03 seconds and 0.05 seconds, respectively. Thus, the maximum theoretical arrival rate supported by the system is 20 req/sec (i.e., 1 / max [0.03, 0.05]) [41]. The average arrival rate starts at a low value

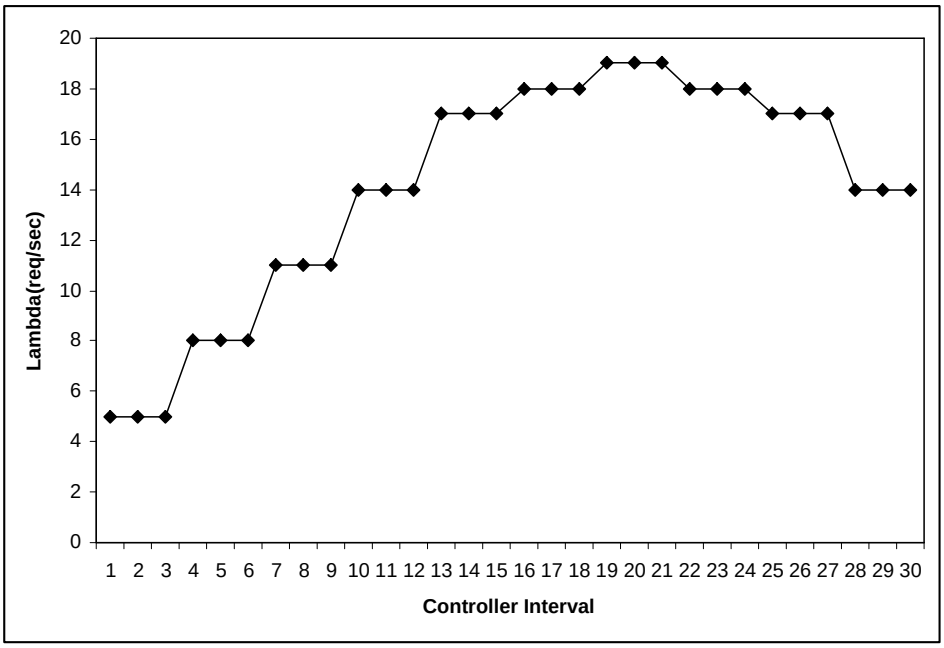
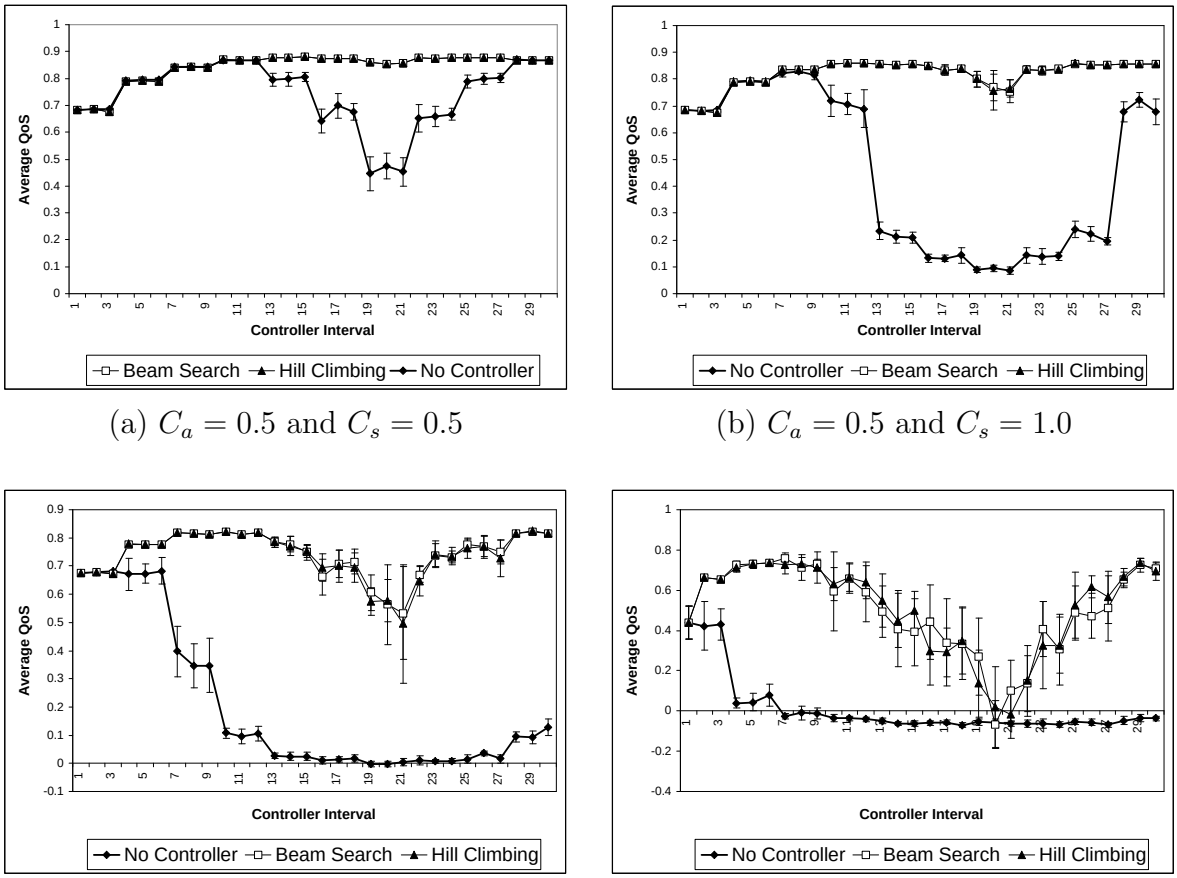


Figure 4.14: Workload intensity variation for the high variability experiments.

of 5 req/sec and reaches a peak of 19 req/sec, close to the theoretical maximum, at CI = 19. The workload intensity stays at this value for three consecutive CIs and then starts to decrease towards 14 req/sec. Ten experiments were run for each combination

of C_a and C_s and 95% confidence intervals for the average of the QoS value were computed at the end of each CI. Results were obtained for three scenarios: one in which the controller is disabled and two others with the QoS controller active. The two results in which the controller is active differ in the combinatorial optimization technique used by the controller: beam search and hill-climbing.



(a) $C_a = 0.5$ and $C_s = 0.5$

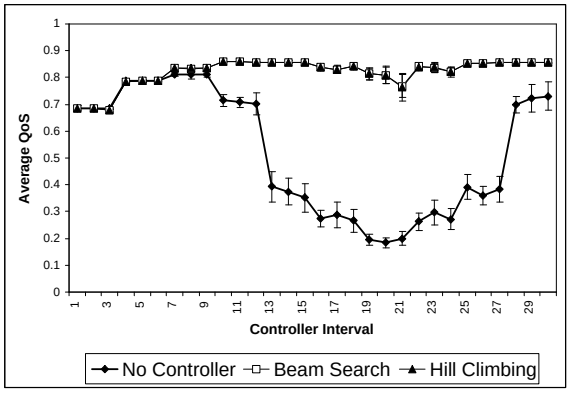
(b) $C_a = 0.5$ and $C_s = 1.0$

(c) $C_a = 0.5$ and $C_s = 2.0$

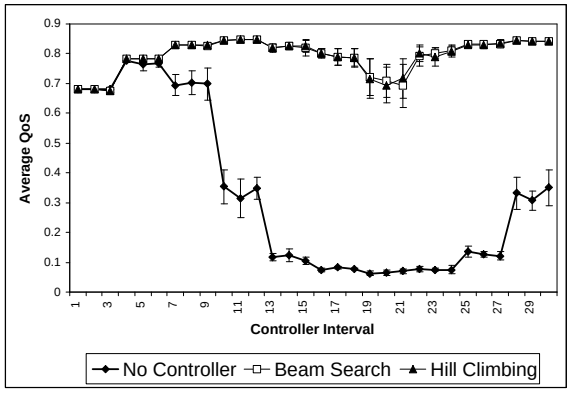
(d) $C_a = 0.5$ and $C_s = 4.0$

Figure 4.15: QoS Controller Performance vs. C_a and C_s ($C_a = 0.5$).

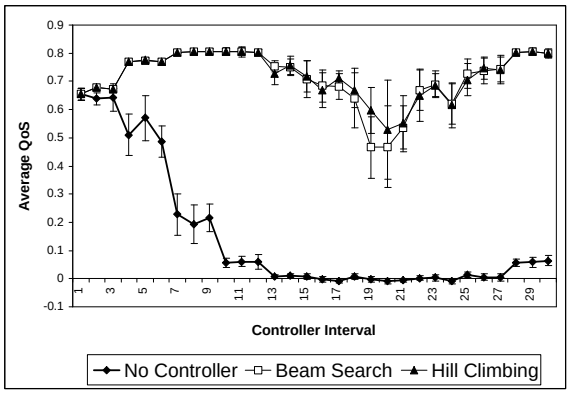
Figures 4.15, 4.16, 4.17, and 4.18 show the results for all the scenarios described above. First, it should be noted that the controlled system always behaves



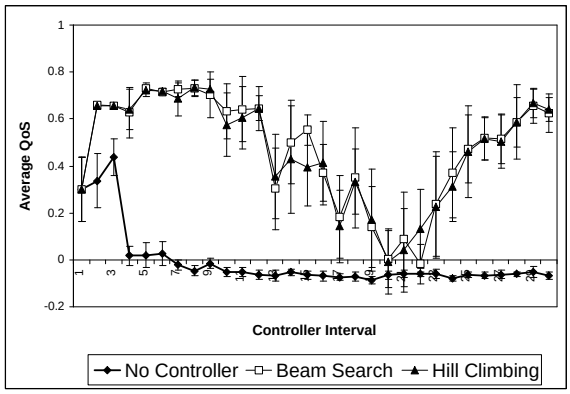
(a) $C_a = 1.0$ and $C_s = 0.5$



(b) $C_a = 1.0$ and $C_s = 1.0$



(c) $C_a = 1.0$ and $C_s = 2.0$

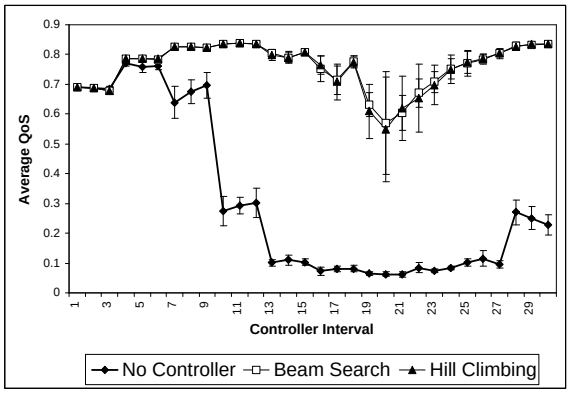


(d) $C_a = 1.0$ and $C_s = 4.0$

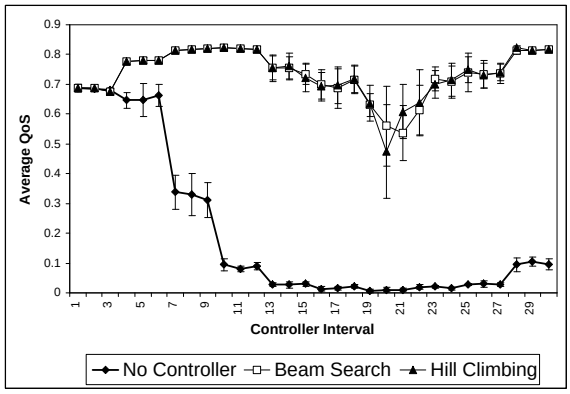
Figure 4.16: QoS Controller Performance vs. C_a and C_s ($C_a = 1.0$).

better, i.e., produces higher QoS values than the non-controlled (NC) system. Also, as expected, confidence intervals become wider as either or both COV increase. But, confidence intervals for the controlled system tend to be wider than those for the non-controlled system (NC) because the system itself is varying due to dynamic adjustment of parameters. Another clear observation is that as the variability increases, the performance of the NC system starts to deviate from that of the controlled systems at an earlier stage. For example, when $C_a = 0.5$ and $C_s = 0.5$ (Fig. 4.15 (a)), the difference in QoS starts at CI = 13 ($\lambda = 14$) req/sec. As C_s increases for the same value of C_a, the difference between the two cases becomes apparent at CI = 10, CI = 5, and CI = 3 (see Figs. 4.15 (b)-(d)).

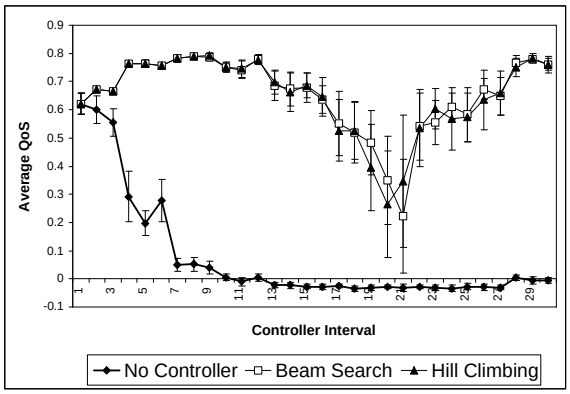
Let us now examine the effect of the variation of C_s for a fixed value of C_a. For $C_a = 0.5$ and $C_s = 0.5$ and $C_s = 1.0$ (Figs. 4.15 (b)-(d)), the controlled system keeps the QoS value close to 0.9 throughout the experiments while the NC system exhibits a marked drop in QoS (to about 0.1 when $C_s = 1.0$) when λ reaches its peak value. For these two values of C_s the QoS for the NC case is still positive. When C_s increases to 2.0 (Fig. 4.15 (c)), the QoS for the controlled case drops to about 0.5 at the peak value of λ and the QoS for the non-controlled case goes to zero for most of the experiment ($13 < \text{CI} < 27$). For $C_s = 4.0$ (Fig. 4.15 (d)), a high value of the service time COV, the NC case exhibits a negative QoS for most of the experiment while the controlled system only approaches zero (the average is above zero but the confidence interval includes zero) at peak load and then recovers. The NC system remains in negative territory. We now examine the variation of the QoS as C_a varies for a fixed value of C_s. For $C_s = 0.5$ and $C_a = 0.5, 1.0$ and 2.0 (Figs. 4.15 (a), 4.16 (a), and 4.17 (a)), the NC system exhibits marked drops in the QoS value as soon as λ starts



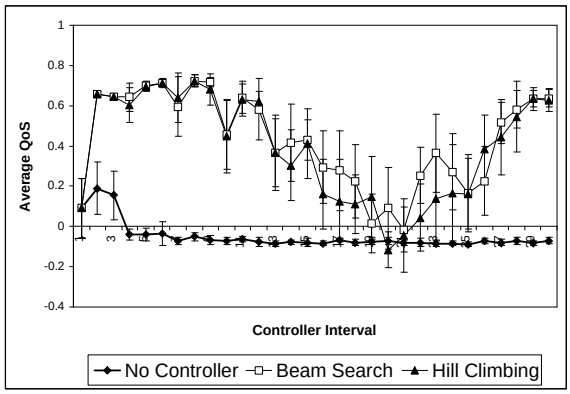
(a) $C_a = 2.0$ and $C_s = 0.5$



(b) $C_a = 2.0$ and $C_s = 1.0$

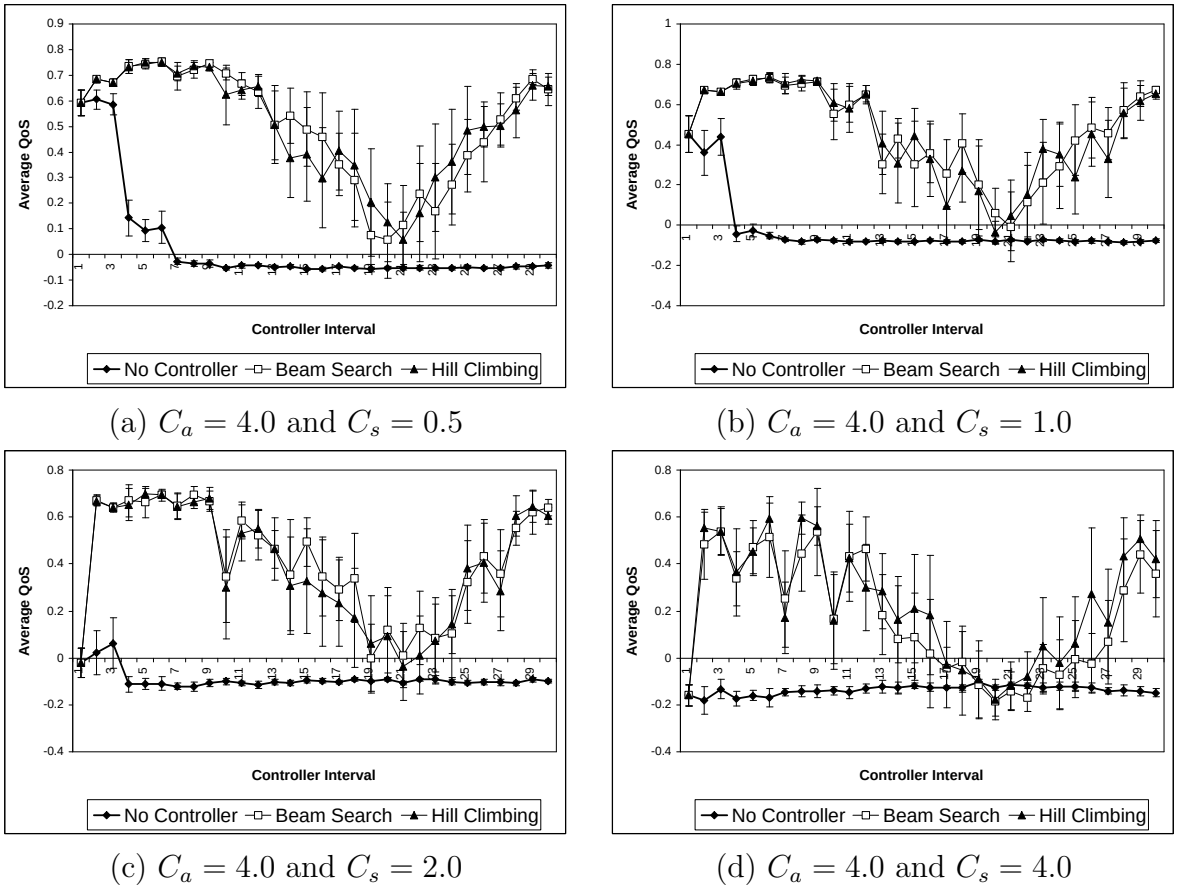


(c) $C_a = 2.0$ and $C_s = 2.0$



(d) $C_a = 2.0$ and $C_s = 4.0$

Figure 4.17: QoS Controller Performance vs. C_a and C_s ($C_a = 2.0$).



(a) $C_a = 4.0$ and $C_s = 0.5$

(b) $C_a = 4.0$ and $C_s = 1.0$

(c) $C_a = 4.0$ and $C_s = 2.0$

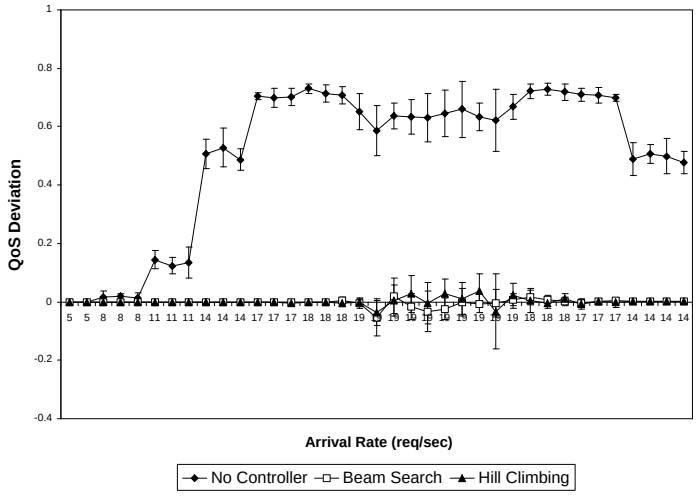
(d) $C_a = 4.0$ and $C_s = 4.0$

Figure 4.18: QoS Controller Performance vs. C_a and C_s ($C_a = 4.0$).

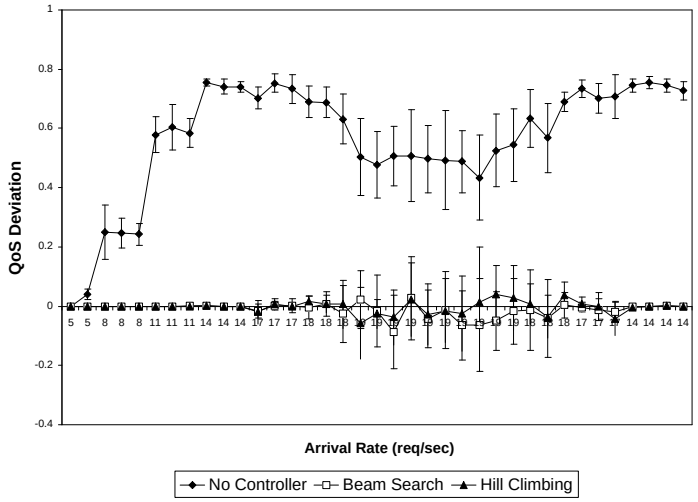
to increase but still remains in positive territory. The controlled system maintains a high QoS value at peak load even for $C_a = 2.0$. For example, in this case, the average QoS value at peak load is 0.6 for the controlled system while it is around 0.1 for the NC system. When $C_s = 0.5$ and $C_a = 4.0$ (Fig. 4.18 (a)), the NC system displays a negative QoS throughout most of the experiment (from CI = 7 onward). The controlled system reaches a small (i.e., 0.1) but positive value of the QoS at peak load. At extreme cases, where both C_a and C_s are very high (i.e., equal to 4.0 as shown in (Fig. 4.18 (d)), the NC system has a negative QoS value throughout the entire experiment. The controlled system reaches some negative points at peak load but recovers when the load decreases.

To complement the observations presented above, we deemed it was necessary to show some additional results that compare the deviation of the QoS with respect to the optimal QoS (obtained from an exhaustive search performed off-line) as C_a and C_s vary. We also stress tested the multithreaded server by elongating the phase of the peak load (19 req/sec) from 3 to 10 controller intervals. Thus, the length of these experiments are 74 minutes (i.e., 37 controller intervals). Figure 4.19 shows the QoS deviation for the case of non-controller, beam search and hill climbing based controllers for $C_a = 1$ and $C_s = 1$, 2, and 4. In Fig. 4.20, we show results that correspond to the cases of ($C_a = C_s = 2$) and ($C_a = C_s = 4$).

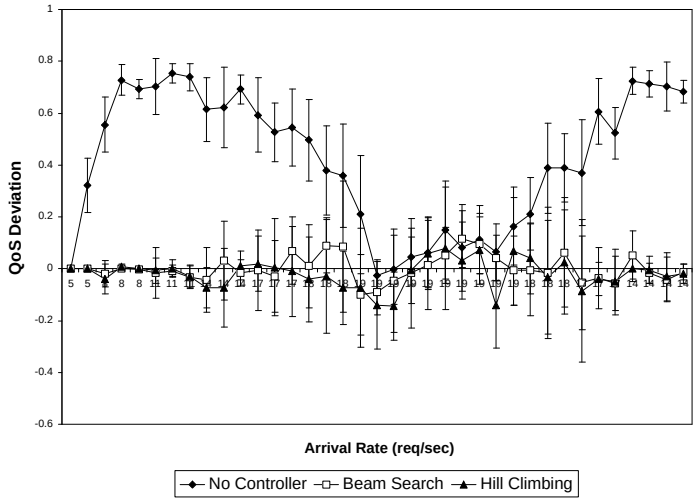
It is worth mentioning that in all these experiments on the robustness of the controller, the initial configuration was chosen in such a manner that it is equally suitable for the controlled and non-controlled systems for the very first controller intervals (i.e., the initial workload). This is reflected in Fig. 4.19-(a) as it shows



(a) $C_a = 1.0$ and $C_s = 1.0$

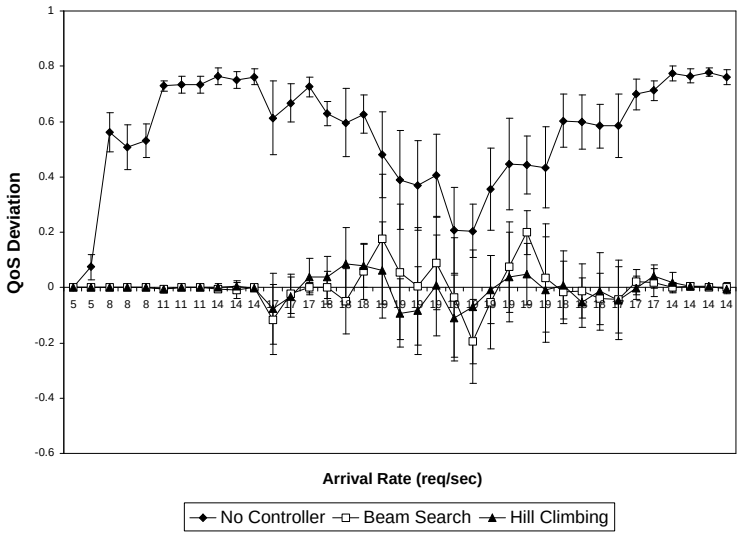


(b) $C_a = 1.0$ and $C_s = 2.0$

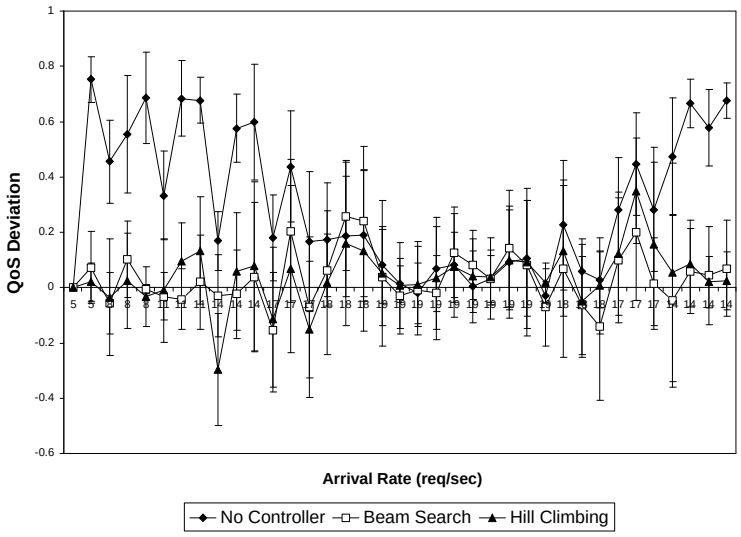


(c) $C_a = 4.0$ and $C_s = 4.0$

Figure 4.19: Deviation of QoS Controller Performance from Optimal vs. C_a and C_s ($C_a = 1.0$).



(a) $C_a = 2.0$ and $C_s = 2.0$



(b) $C_a = 4.0$ and $C_s = 4.0$

Figure 4.20: Deviation of QoS Controller Performance from Optimal for ($C_a = C_s = 2$) and ($C_a = C_s = 4$).

that there is virtually no difference between the controller and no-controller case for the first four controller intervals. This same figure also shows that the QoS deviation from the optimal value for the no-controller case gets higher as the workload intensity increases. On the other hand, the QoS deviation is almost null for both the cases of beam search and hill climbing. However, as C_s increases, the QoS deviation for the no-controller case gets higher for most of the workload intensity except for the peak load where the QoS deviation gets smaller (see Fig. 4.19-(b)-(c)). When $(C_a = 1, C_s = 4)$, the QoS deviation is similar to that of the controller (for both beam search and hill climbing) at peak load (19 req/sec for 10 controller intervals). The reason for that is that the initial configuration happens to be suitable for the peak load phase. The QoS deviation for both beam search and hill climbing gets higher as the workload intensity and C_s increase, but remains very close to the X-axis (the 95% confidence intervals include 0 all the time). These same observations can be made about the case of $(C_a = C_s = 2)$ as illustrated in Fig. 4.20-(a). For the extreme case of $(C_a = C_s = 4)$ (Fig. 4.20-(b)), the QoS deviation gets even wider for both the no-controller and controller cases. However, here again, the QoS deviation is much higher for the non-controlled system except at the peak load where all three systems have similar QoS deviation values.

4.1.7 Sensitivity Analysis

In order to explore the sensitivity of the controller to the space of SLA values, we ran experiments for $C_a = C_s = 2.0$ for stricter and more relaxed SLA values than the ones used in Fig. 4.17 (c). Figure 4.21 illustrates the relative variation φ of the QoS with respect to the base value QoS_{base} shown in Fig. 4.17 (c). The value of φ was

defined as

$$\varphi = \frac{QoS - QoS_{\text{base}}}{\mid QoS_{\text{base}} \mid}. \tag{4.6}$$

The values for the more relaxed and stricter SLAs are: $R_{\max} = 1.5$ seconds, $X_{\min} = 4$

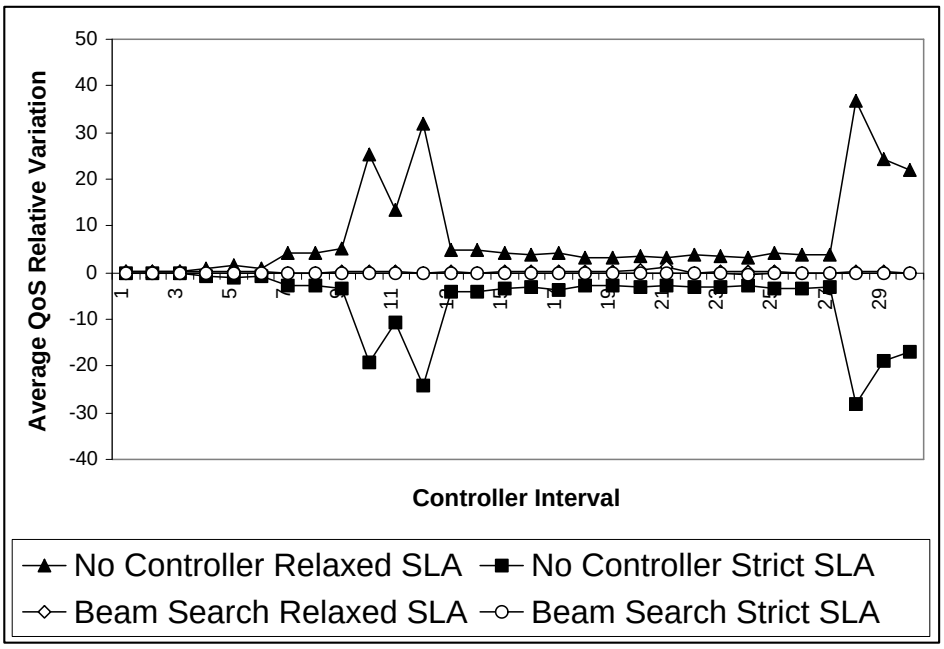


Figure 4.21: Effect of stricter and more relaxed SLAs on the controller performance.

requests/sec, $P_{\max} = 0.1$; and $R_{\max} = 1.0$ seconds, $X_{\min} = 7$ requests/sec, $P_{\max} = 0.03$, respectively. As the figure indicates, the controlled system is much less sensitive to variations in the SLA values than the NC system.

4.1.8 Workload Forecasting

In the self-managing computer systems that we proposed in the previous section and in [46], the QoS optimizer module uses the average arrival rate of requests obtained

in the previous controller interval, CI, as an estimate of the expected workload intensity for the next CI. This value is then used by the performance model to compute the QoS value for a given set of configuration parameters. The drawback of such an approach is that it overlooks any increase or decrease tendency in the workload for the past CI. This could result, consequently, in a very inaccurate estimate of the next expected arrival rate and an inappropriate choice of configuration values. To overcome this shortcoming, we added a module responsible for short-term workload forecasting. This module keeps a sliding window of N values for the last average arrival rates observed for the last N small sub-intervals. Each of these sub-intervals is of length Δ seconds. N and Δ are chosen so that $N \times \Delta$ does not exceed the length of a controller interval (2 minutes in our case). Many techniques can be used for short-term forecasting. However, no particular technique gives good forecasting results for any kind of data. Therefore, the forecasting module uses the three techniques presented in Chapter 2, namely: exponential smoothing, weighted moving averages, and polynomial regression [40]. The choice of these particular forecasting techniques is motivated by the nice properties that they have and that are discussed in section 2.1.3. Exponential smoothing computes a prediction as follows: PredictedValue = $\alpha\times$ PreviousActualValue + $(1-\alpha)\times$ PreviousPredictedValue. In our experiments we used $\alpha = 0.6$. For weighted moving averages is an appropriate technique for these situations. In our experiments, we compute the forecast value based on the three most recent average arrival rates in the sliding window. The chosen weights give more importance to the newest values. Hence, the forecast value is given by: ForecastValue = (0.45 $\times$ LatestEntryInSlidingWindow + 0.35 $\times$ SecondLatestEntryInSlidingWindow + 0.20 $\times$ ThirdLatestEntryInSlidingWindow). For polynomial regression, we used a

moderately high value for polynomial degree: six. All three models are rebuilt each time a new average arrival rate entry is inserted into the sliding window. At this time, we compute what would be the forecast value according to each of the three models. We also compute the R^2 value, based on the method of the least squares errors, for each of these models to assess the quality of the fits. At this stage, the forecasting module returns the forecast value provided by the model with the highest R^2 value. There is an exception to this rule, however. In the case of a downward trend in the workload intensity, the polynomial regression model may forecast a negative value for the expected arrival rate. In such a case, even though the polynomial regression model might produce the highest R^2 value, the forecasting module returns the forecast value that comes from the model with the second highest R^2 value, instead. Clearly, our design for the workload forecasting module makes it easily extendable to include additional forecasting techniques that are suitable for on-line analysis.

4.1.8.1 Experimental Setting

The experimental setting used in this section is very similar to that used in the previous section with the following modifications:

- The inter-arrival time and service time are exponentially distributed (i.e., $C_a = C_s = 1$).
- Beam search is the only heuristic search technique used since the results of the previous section show that there is no statistically significant difference at the 95% level between beam search and hill climbing.
- The IMSL library [28] is used for the polynomial regression models needed by

the forecasting module.

- Since this section uses a different workload than that of the previous section, different initial configurations are needed (so as not to purposefully disadvantage the non controlled system). Thus, in this experiment the initial values for n and m are 30 and 10, respectively.

The SLA values and their corresponding weights are still the same:

- $R_{\max} = 1.2$ seconds, $w_R = 0.35$,
- $X_{\min} = 5$ requests/sec, $w_X = 0.25$, and
- $P_{\max} = 0.05$, and $w_P = 0.4$.

4.1.8.2 Results

Figure 4.22 compares the expected arrival rate at every controller interval, when the forecasting module was enabled/disabled, to the actual measured arrival rate. Note that in this figure we start from the 2nd controller interval as it is only at this time that data is available in the sliding window so that forecasting can be carried out. The actual workload has two peaks: 30 req/sec at CI = 8 and CI = 24. The curve for the expected arrival rate when forecasting is not used is simply a one-time unit shift to the right of the curve of the measured arrival rate. When the forecasting module is enabled, the system succeeded in finding quite close estimates of the arrival rate when that was possible at all. The largest gaps between the forecast and the measured arrival rates happened at the 10-th and 11-th controller intervals. At these points, the forecast values were 41.27 req/sec and 43.28 req/sec, whereas the measured workload

intensities were 30 req/sec and 26.03 req/sec, respectively. However, since for both of these cases, the measured arrival rates for the immediate previous controller intervals (9 and 10) were 30 req/sec, these gaps did not significantly impact the QoS. This is due to the fact that 30 req/sec exceeds by far the system's maximum throughput (20 req/sec). Therefore, the system configuration was already set at its minimum size.

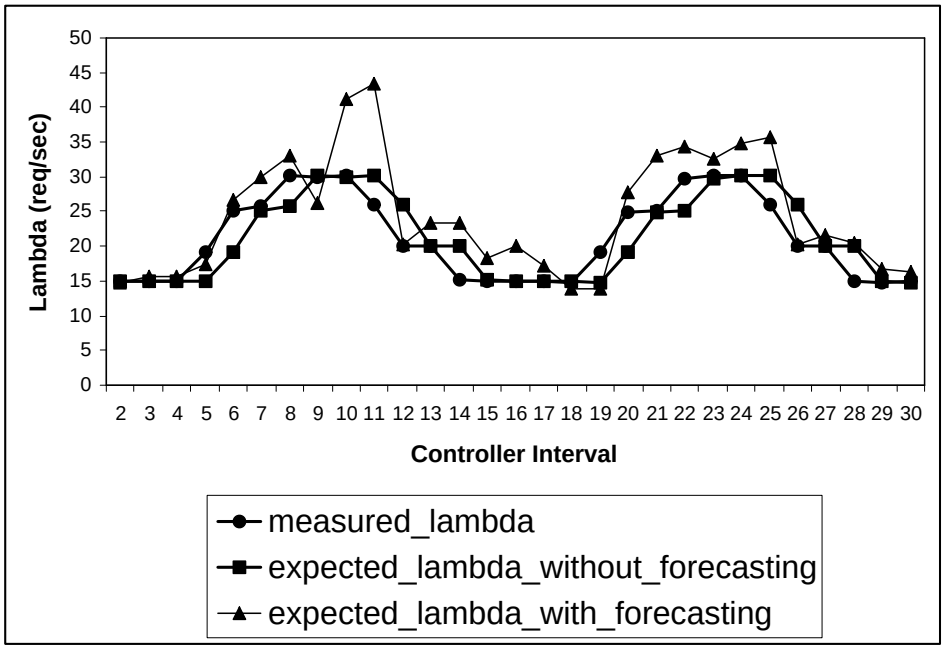


Figure 4.22: Workload intensity variation for the workload forecasting experiments.

Figure 4.23 shows the results of the average QoS obtained for 10 runs of the simulation when the forecasting module was enabled and when it was disabled along with the 95% confidence intervals for the average QoS. We can see from this figure that the average QoS obtained when forecasting is enabled is statistically slightly better for exactly 8 out the 30 controller intervals. For the other controller intervals the 95% confidence intervals overlap and therefore no conclusion can be made. These eight controller intervals are: CI = 6 (λ = 25 req/sec), CI = 7 (λ = 26 req/sec), CI =

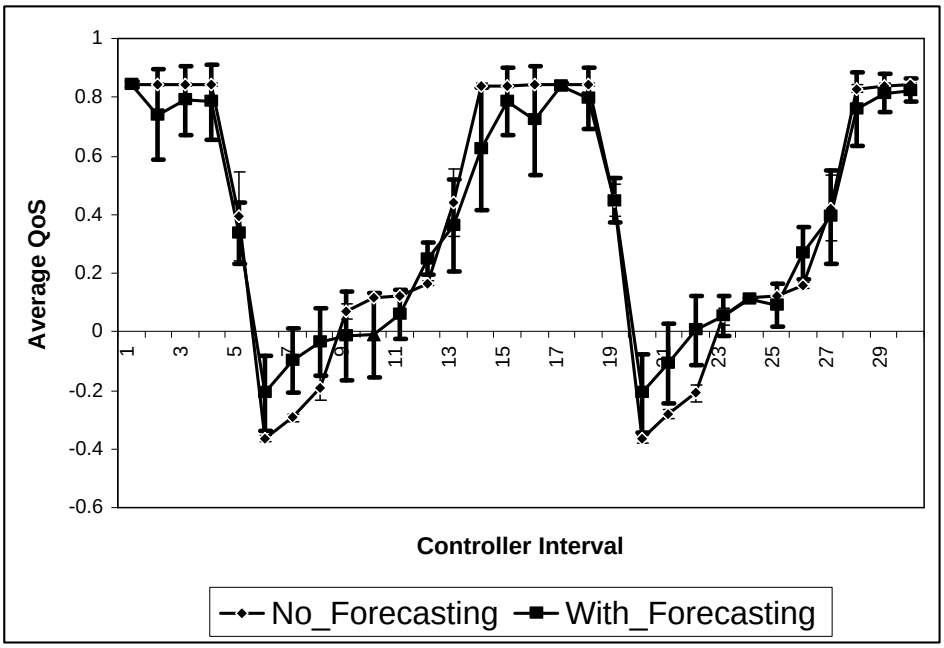


Figure 4.23: Impact of workload forecasting.

8 (λ = 30 req/sec), CI = 12 (λ = 20 req/sec), CI = 20 (λ = 24.92 req/sec), CI = 21 (λ = 25.2 req/sec), CI = 22 (λ = 29.74 req/sec), and CI = 26 (λ = 20 req/sec). For most of these controller intervals, the QoS is negative. However, when forecasting is enabled, the QoS values are significantly higher than otherwise. For example, at the 6-th controller interval, the average QoS when forecasting is not used is -0.36 whereas it is only -0.20 if forecasting is used. This is an improvement of about 44%. The forecasting module was able to notice that λ went up from 14.91 req/sec at CI = 4 to 19.12 req/sec at CI =5 and therefore predicted a value of 26.66 req/sec for CI = 6. The actual measured value of λ was 25 req/sec. Another scenario that shows the importance of the added forecasting module is the measured QoS at the 26-th controller interval (λ = 20 req/sec). The measured QoS is 0.27 when forecasting is enabled and only 0.16 when it is disabled. This is an improvement of about 69%.

The forecasting module noticed that λ went down from 30 req/sec at CI = 24 to 26 req/sec at CI =25 and predicted a value of 20 req/sec for CI = 26. The actual measured value for λ for CI = 26 is exactly 20 req/sec.

It is worth mentioning, however, that significantly higher improvements are attained only when workload forecasting is used in combination with a dynamic frequency of control as shown in the next section.

4.1.9 Frequency of Control

In this section we investigate the impact of the frequency of control on the overall performance of the controller in terms of the measured QoS. Before we present an adaptive controller algorithm that is used to dynamically regulate the frequency at which the controller is invoked, we make the following remark. For reasons of clarity, we introduce a new term: monitoring interval. The term controller interval keeps its same meaning as in the previous sections. Controller interval still refers to the elapsed time between two consecutive executions of the controller algorithm. Monitoring intervals, on the other hand, refers to the time interval used for aggregating the different measured performance metrics. Monitoring intervals have a fixed size during the entire experiment (2 minutes in our case). In previous sections, all controller intervals also had a fixed size (2 minutes) and hence were equivalent to monitoring intervals. In this section, however, controller intervals are of an arbitrary size that is dynamically determined by the following proposed algorithm. Figure 4.24 shows an algorithm that can be used to dynamically vary the length of the control interval. This algorithm sets the length of the CI as a multiple, K, of the smallest possible control interval CI_{min}. When the currently measured value of the QoS, QoS_{curr}, is

less than or equal to a minimum value $\mathrm{QoS}_{\mathrm{min}}$, the controller interval is set to its minimum value $\mathrm{CI}_{\mathrm{min}}$. Otherwise, the controller interval is set to a multiple of $\mathrm{CI}_{\mathrm{min}}$ according to the relative error ϵ between the QoS value, $\mathrm{QoS}_{\mathrm{prev}}$, measured last time the controller was activated and the currently measured value of the QoS, $\mathrm{QoS}_{\mathrm{curr}}$.

If $\mathrm{QoS}_{\mathrm{curr}} < \mathrm{QoS}_{\mathrm{min}}$
then $CI \leftarrow \mathrm{CI}_{\mathrm{min}}$
else begin
$\epsilon = \left| \frac{\mathrm{QoS}_{\mathrm{curr}} - \mathrm{QoS}_{\mathrm{prev}}}{\mathrm{QoS}_{\mathrm{prev}}} \right|$
If $0 \leq \epsilon \leq 0.05$ then $K = 12$
If $0.05 < \epsilon \leq 0.1$ then $K = 6$
If $0.1 < \epsilon \leq 0.2$ then $K = 5$
If $0.2 < \epsilon \leq 0.3$ then $K = 4$
If $0.3 < \epsilon \leq 0.4$ then $K = 3$
If $0.4 < \epsilon \leq 0.5$ then $K = 2$
If $\epsilon > 0.5$ then $K = 1$
$CI \leftarrow K \times \mathrm{CI}_{\mathrm{min}}$
end

Figure 4.24: Algorithm for adjusting control interval length

Figure 4.25 shows the variation of the QoS when the control interval varies according to the algorithm of Fig. 4.24 when $C_a = C_s = 1.0$. In these curves, workload forecasting is always used. The workload used in that experiment has two peaks: one at time 6 and another at time 20. It can be clearly seen from the figure that the use of a dynamically adjusted controller interval yields better QoS values. For example, at peak loads (see monitoring intervals 6 and 20), the QoS for the dynamically adjusted system is always positive. These curves also show that the QoS values obtained when adaptive control intervals are used are generally higher than those achieved when the controller runs at fixed intervals.

One could ask the question whether these improvements come mainly from the dynamic adjustment of the control interval and not because of workload forecasting.

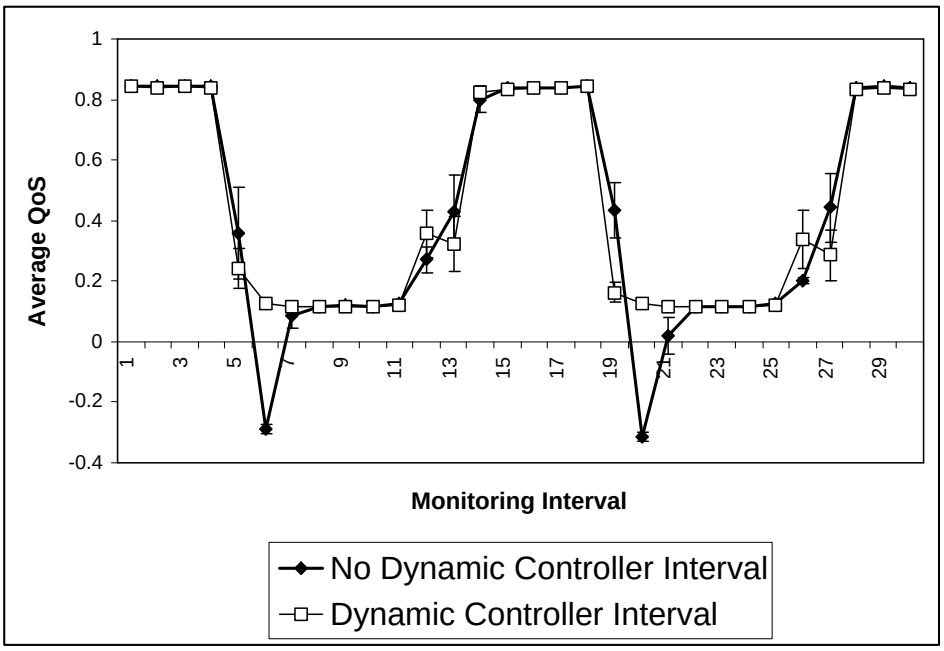


Figure 4.25: Dynamic controller interval impact on QoS (forecasting always used)

To this end, we conducted another set of experiments in which we compare the average QoS values obtained for the cases when dynamic controller intervals were used alone against the cases when they were used jointly with workload forecasting. The results are reported in Fig. 4.26 for $C_a = C_s = 1.0$. The curves in this figure clearly show that there is a statistically significant performance gain when forecasting is enabled in conjunction with dynamic controller intervals. There is an accompanying increase in the QoS gain as a result of using dynamic controller intervals combined with workload forecasting.

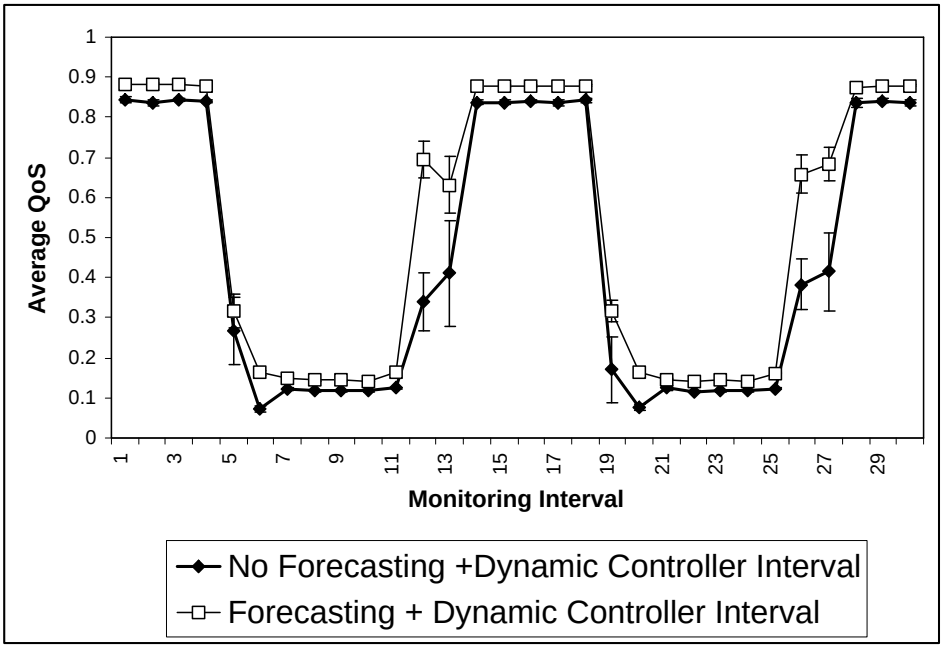


Figure 4.26: Impact of workload forecasting on QoS (dynamic controller intervals always used)

4.2 Illustration of the Control Approach on a Multiple Class Multithreaded Server

In this section we investigate the efficiency of the control approach for the case of a multiple class multithreaded server. The goal of this section is to show through the simulation of a multithreded server with multiple classes of requests that our suggested performance model presented in section 4.2.2 achieves the following:

- The controlled multiple class multithreaded server yields a higher QoS than the non controlled server.
- The controller strives to adhere to respecting the relative importance (priority) of the different classes as expressed in their SLAs.

4.2.1 System Description

The multithreaded server services C distinct classes of customer requests. The classes are numbered from $c = 1, 2, ..., C$. All arriving requests to the system join the same single queue of incoming requests that wait for an avaialable thread to start working on them. A thread in the server picks the request that is at the front of the queue regardless of its class. In other words, a thread can service any class of requests. Once a thread begins working on a request, it starts competing for the physical resources with the other threads who are servicing other requests possibly from other classes of customers. After a request gets serviced at the CPU and the disk, it departs from the system. The server has only a finite queue of requests and a finite number of threads. Therefore, a request that arrives to the server and finds no room in the waiting queue for available threads is rejected. The inter-arrival times for each class of requests are

assumed to be exponentially distributed as well as the requests service times at the CPU and the disk.

Each class, c, has its own SLAs expressed in terms of maximum response time and maximum probability of rejection. There are also weights associated with the response time and the probability of rejection of requests for each class c. These are denoted by w_{R_c} and w_{Prej_c}, respectively and they satisfy the following constraints:

- $\sum_{c=1}^{C} w_{R_c} = 1$, and
- $\sum_{c=1}^{C} w_{Prej_c} = 1$.

The global QoS, at configuration vector $\vec{C}$ and workload vector $\vec{W}$, is defined then as: $QoS(\vec{C}, \vec{W}) = W_R * QoS_R(\vec{C}, \vec{W}) + W_{Prej} * QoS_{Prej}(\vec{C}, \vec{W})$ where:

- $W_R + W_{Prej} = 1$, and
- $QoS_R(\vec{C}, \vec{W}) = \sum_{c=1}^{C} w_{R_c} * \Delta QoS_{R_c}(\vec{C}, \vec{W})$, and
- $QoS_{Prej}(\vec{C}, \vec{W}) = \sum_{c=1}^{C} w_{Prej_c} * \Delta QoS_{Prej_c}(\vec{C}, \vec{W})$.

$\Delta QoS_{R_c}(\vec{C}, \vec{W})$ and $\Delta QoS_{Prej_c}(\vec{C}, \vec{W})$ are, respectively, the relative deviations of the response time and probability of rejection for class c with respect to its SLAs for configuration vector $\vec{C}$ and workload vector $\vec{W}$. Their exact formulas are given in section 3.1.2.

4.2.2 The Multiple Class Performance Model

The multiclass performance algorithm used by the controller has two phases. The first phase consists of the initialization of the average number of requests in the execution

mix for every class. The first phase is given in Fig. 4.27.The second phase of the algorithm consists of the necessary iterative steps that need to be carried out before the algoritm actually converges to the solution and that performance metrics can be obtained. The second phase is given in Fig. 4.28.

It is worth noting that this algorithm is adapted from the generalized algorithm that appears in [37]. In our case, all the classes are memory constrained. The following notation is used in the algorithm:

- I: Number of resources.
- C: Number of classes.
- i: index for resources.
- c: index for classes.
- J: Number of threads in the system.
- K: System size (queue length + number of threads in system).
- $\bar{n}_{c,system}$: Number of class c requests in the system.
- $\bar{n}_c$: Number of class c requests in execution.
- FESC: Flow Equivalent Service Center.

For every device i and class c **Do**:
Begin
$U_{i,c}(\vec{\lambda}) = \lambda_c * D_{i,c}$
$U_i(\vec{\lambda}) = \sum_{c=1}^{C} U_{i,c}(\vec{\lambda})$
If $(U_i(\vec{\lambda}) < 1.0)$ for all i **Do**:
$\bar{n}_{i,c}(\vec{\lambda}) = \frac{U_{i,c}(\vec{\lambda})}{1-U_i(\vec{\lambda})}$
For every class c **Do**:
set $\bar{n}_c$ as: $\bar{n}_c(\vec{\lambda}) = \sum_{i=1}^{I} \bar{n}_{i,c}(\vec{\lambda})$
adjust $\bar{n}_c$ as: $\bar{n}_c(\vec{\lambda}) = min\{\bar{n}_c(\vec{\lambda}), J_c\}$ where: $J_c = J * \frac{\bar{n}_c(\vec{\lambda})}{\sum_{c=1}^{C} \bar{n}_c(\vec{\lambda})}$
Else
use an alternative initialization:
For every class c set:
$\bar{n}_c(\vec{\lambda}) = \frac{\lambda_c}{\sum_{r=1}^{r=C} \lambda_r}$ and $\bar{n}_{c,system}(\vec{\lambda}) = \bar{n}_c(\vec{\lambda})$
End

Figure 4.27: Initialization Phase of the Performance Model for a Multiclass Multithreaded Server.

4.2.3 Case of a Simulated Multithreaded Server With Multiple Classes of Requests

In this section we report the results obtained when using the proposed performance model for multiple classes of requests in section 4.2.2 to control a simulated multithreaded server with several classes of workloads.

4.2.3.1 The Experimental Setting

In the experiments reported in this section we consider a multithreaded server with 3 classes of requests. Table 4.2 shows the input parameters considered for these experiments. The table shows the service demands in seconds for each class, the maximum response time (in seconds) and the maximum probability of rejection for each class, and the weights for the response time as well as the weights for the probability of

Table 4.2: Input Parameters for the Experiments

Multiclass Multithreaded Server			
C	3		
c	1	2	3
$D_{CPU,c}$	0.006	0.005	0.004
$D_{disk,c}$	0.004	0.005	0.006
R_{max_c}	0.15	0.15	0.15
$Prej_{max_c}$	0.01	0.01	0.01
w_{R_c}	0.35	0.35	0.30
w_{Prej_c}	0.35	0.35	0.30

rejection for each class(w_{R_c} and w_{Prej_c} values, respectively).

Initially, $W_R = 0.40$ and $W_{Prej} = 0.60$. Therefore, a higher importance is given to the overall probability of rejection of requests. The initial configuration for the experiments is 10 threads and a system size equal to 30.

4.2.3.2 Results

Ten runs were conducted for the experiments and 95% confidence intervals were obtained for the results reported in this section. The workloads for the 3 classes were varied as shown in Fig. 4.29 for these experiments.

Figure 4.30 shows the variation of the global QoS function during the experiment for the controller and non-controller cases. The graph also shows the 95% confidence intervals. The graph clearly indicates that the global QoS function is significantly higher when the controller is used and that it is always positive. For the non-controller case, however, the QoS is in the negative territory between controller intervals 6 and 23. This indicates that at least one class has seen a violation of at least one of its SLAs (maximum response time or maximum probability of rejection). As shown in

Fig. 4.29, the workloads intensity slightly increases between controller intervals 6 and 23 before dropping back again to initial levels.

Figure 4.31 shows the variation of the average response time for the three workload classes for the controller and non-controller cases. As it can be seen in this figure, the average response time for the controller case is significantly higher than in the non-controller case between controller intervals 6 an 23. At several occasions, the average response time for the controller case goes beyond the target maximum response time of 0.15 sec with the highest response time of 0.27 sec occuring at CI = 15. For the non-controller case, on the other hand, the average response time is always below its SLA.

This counter-intuitive result is best explained when we look at Fig. 4.32 that corresponds to the variations of the probabilities of rejection. This figure shows that, unlike for the case of the response times, the probabilities of rejection for the case of the controller are much lower than in the non-controller case. For the non-controller case, the probabilities of rejection go beyond their SLA of a maximum probability of rejection of 0.01 between controller intervals 6 and 23. For the controller case, the probabilities of rejection are below their SLA for the entire experiment and their highest value is 0.0016 only.

The explanation of this controller behavior is that because $W_{Prej} > W_R$, the controller realized that some sacrifices in terms of response times at some points would be compensated for by even greater gains in terms of probabilities of rejection and that would ultimately result in an overall superior global QoS.

To verify this claim that the controller behaves well in accordance to the overall importance given to either the response time or the probability of rejection factors,

we conducted an additional set of experiments where we switched the values of W_{Prej} and W_R. Thus, we ran experiments under the same conditions as before (including the same workloads variations) but with $W_{Prej} = 0.40$ and $W_R = 0.60$. This case is referred to as R-higher-weight (for "R has higher weight"). The preceeding case corresponded, therefore, to a Prej-higher-weight (for "Prej has higher weight").

Figure 4.33 shows the variation of the global QoS function during the R-higher-weight experiment for the controller and non-controller cases. The graph also shows the 95% confidence intervals. As expected, the graph shows that the global QoS function is significantly higher when the controller is used and is always positive.

It is more importantly related to our contoller behavior claim to compare the response times and the probabilities of rejections under the controlled mode for the 2 cases of R-higher-weight and Prej-higher-weight.

Figure 4.34 compares the response time for class 1 for the case of R-higher-weight and Prej-higher-weight when the controller is always enabled. This figure shows that for the case of R-higher-weight the achieved response times are generally lower than in the case of Prej-higher-weight. For example, the largest measured average response time at CI = 15 was reduced from 0.27 sec to 0.20 sec.

Likewise, Fig. 4.35 compares the probability of rejection for class 1 for the case of R-higher-weight and Prej-higher-weight when the controller is always enabled. This figure shows that for the case of Prej-higher-weight the achieved probabilities of rejection are generally lower than in the case of R-higher-weight. For example, the largest measured probability of rejection at CI = 16 was dropped from 0.003 to zero.

Similar results apply to classes 2 and 3 as well. These results indeed show that the controller is flexible enough that its behavior is driven by the relative importance

given to the different performance metrics of interest.

Step 1:
For every class $c = 1, 2, ..., C$ **Do**:
Begin
Set $\delta_c = \sum_{c' \neq c} \bar{n}_{c'}(\vec{\lambda})$
Set $\gamma_c = \sum_{c' \neq c} \bar{n}_{c',system}(\vec{\lambda})$
For $n_c = 1, 2, ..., \lfloor J - \delta_c \rfloor$ **Do**:
Call aMVA to compute $X_c(\bar{n}_1, \bar{n}_2, ..., n_c, ..., \bar{n}_C)$
If $(J - \delta_c)$ is not an integer:
Call aMVA to compute $X_c(\bar{n}_1, \bar{n}_2, ..., J - \delta_c, ..., \bar{n}_C)$
Define a single class load dependent model M/M/J/K
with the following throughputs:
$\mu_c(n) = X_c(n)$, for $1 \leq n \leq J - \delta_c$, and
$\mu_c(n) = X_c(J - \delta_c)$, for $J - \delta_c < n \leq \lfloor K - \gamma_c \rfloor$.
Compute the queue length distribution, P_i, as:
(from a single class M/M/J/K model)
$P_i = P_0 \times \frac{\lambda_c^i}{\beta(i)}$, for $1 \leq i \leq \lfloor J - \delta_c \rfloor$, and
$P_i = P_0 \times \frac{\rho^i \times \phi^{\lfloor J - \delta_c \rfloor}}{\beta(\lfloor J - \delta_c \rfloor)}$, for $\lfloor J - \delta_c \rfloor + 1 \leq i \leq \lfloor K - \gamma_c \rfloor$
where $\phi = \mu_c(J - \delta_c)$, $\rho = \lambda_c / \phi$, $\beta(i) = \prod_{j=1}^{j=i} \mu_c(j)$, and
$P_0 = [1 + \sum_{j=1}^{\lfloor J - \delta_c \rfloor} \lambda_c^j / \beta(j) + \frac{\rho \times \lambda_c^{\lfloor J - \delta_c \rfloor} \times (1 - \rho^{\lfloor K - \gamma_c \rfloor - \lfloor J - \delta_c \rfloor})}{\beta(\lfloor J - \delta_c \rfloor) \times (1 - \rho)}]^{-1}$
Compute the performance metrics as:
average number of requests in system:
$\bar{n}_{c,system} = \sum_{i=1}^{i=\lfloor K - \gamma_c \rfloor} (i \times P_i)$
average throughput:
$X_{0,c}(\vec{\lambda}) = \sum_{j=1}^{j=\lfloor J - \delta_c \rfloor} P_j \times X_c(j) + X_c(J - \delta_c) \times [1 - \sum_{j=0}^{j=\lfloor J - \delta_c \rfloor} P_j]$
probability of rejection:
$P_{c,Rej}(\vec{\lambda}) = P_{\lfloor K - \gamma_c \rfloor}$
average response time:
$R_{0,c}(\vec{\lambda}) = \frac{\bar{n}_{c,system}}{X_{0,c}(\vec{\lambda})}$
average number of requests in execution:
$\bar{n}_c = \sum_{j=1}^{j=\lfloor J - \delta_c \rfloor} j \times P_j + (J - \delta_c) \times [1 - \sum_{j=0}^{j=\lfloor J - \delta_c \rfloor} P_j]$
End
Step 2: Repeat step 1 until successive values for $\bar{n}_c$ are sufficiently close for all classes.
Step 3: Return the performance metrics as computed in the last iteration of Step 1.
End.

Figure 4.28: Iterative Phase of the Performance Model for a Multiclass Multithreaded Server.

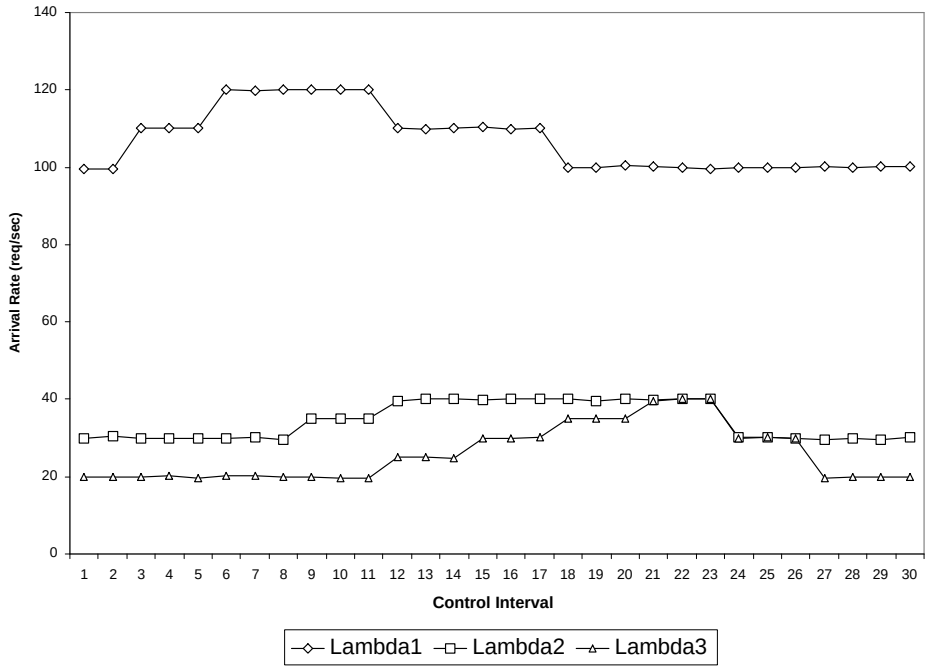


Figure 4.29: Variation of the workload intensity for classes 1, 2, and 3.

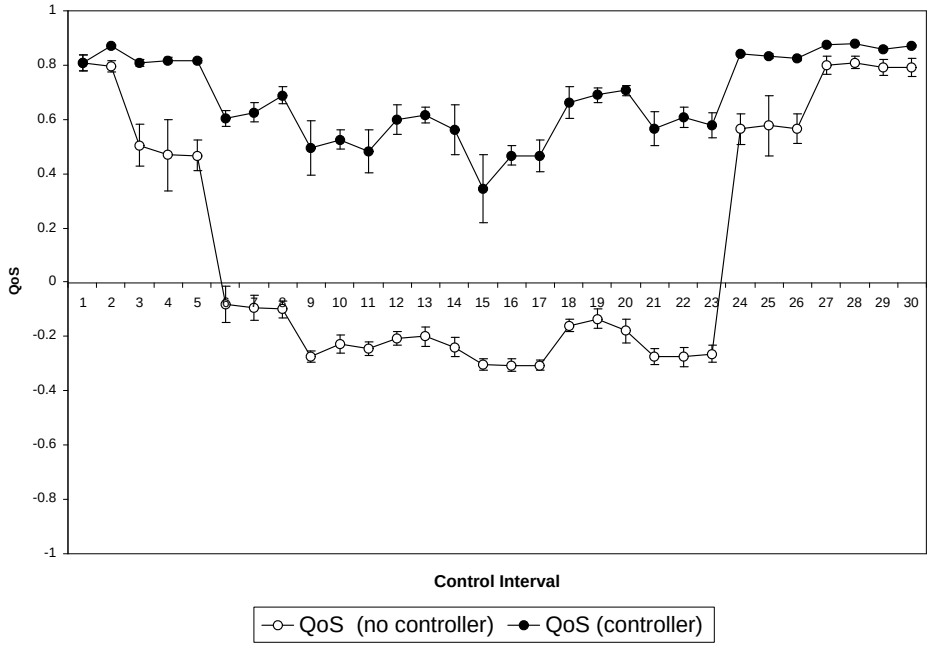


Figure 4.30: Variation of the global QoS function.

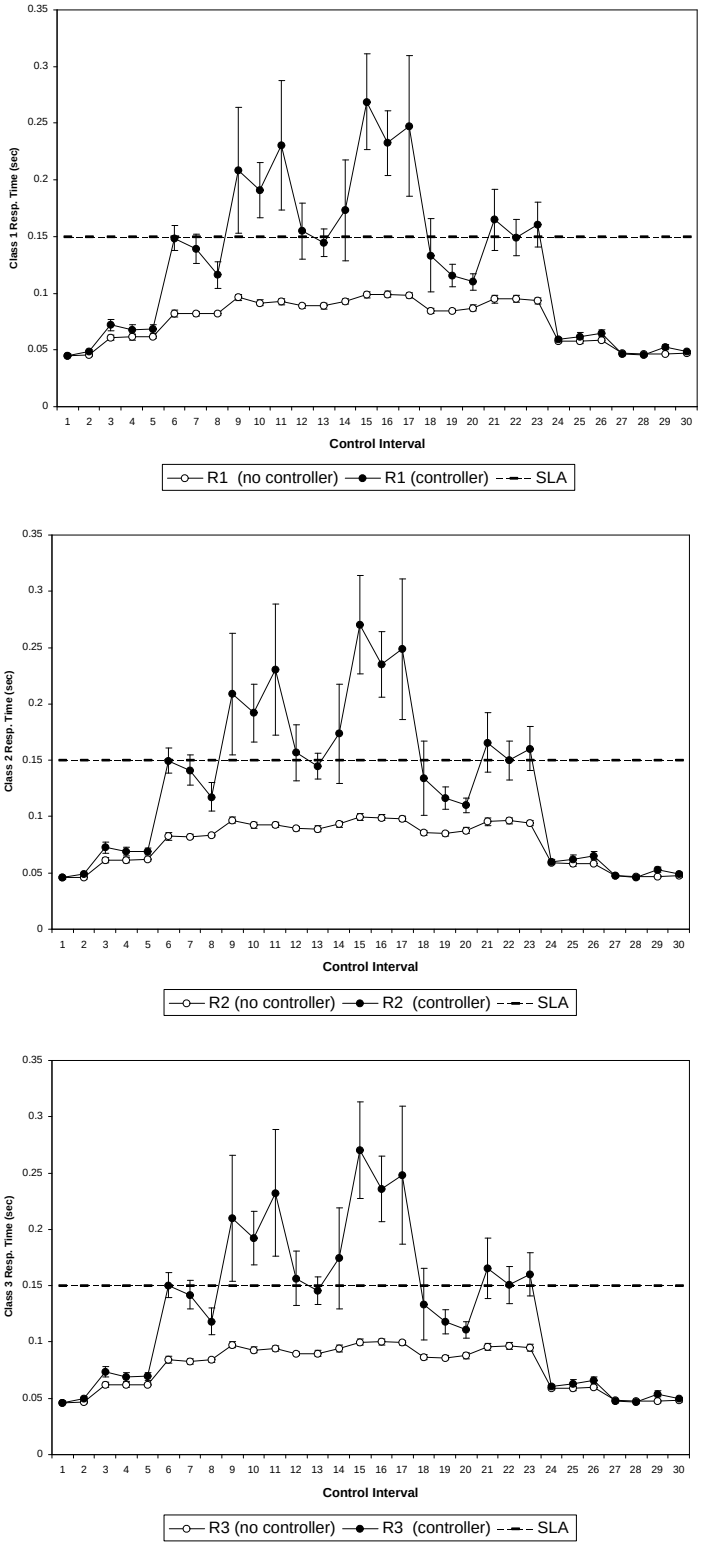


Figure 4.31: Variation of the average response times R_1 (top), R_2 (middle), and R_3 (bottom).

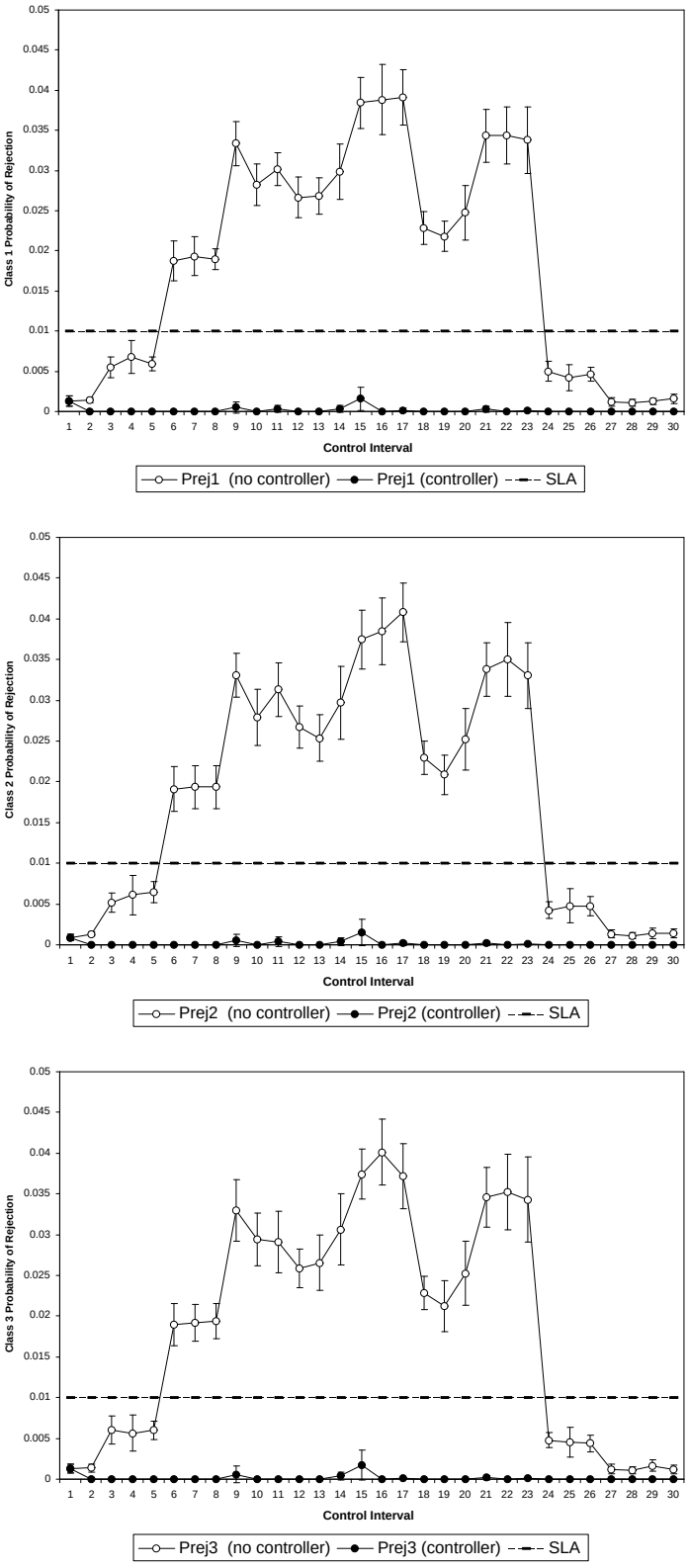


Figure 4.32: Variation of the probabilities of rejection $Prej_1$ (top), $Prej_2$ (middle), and $Prej_3$ (bottom).

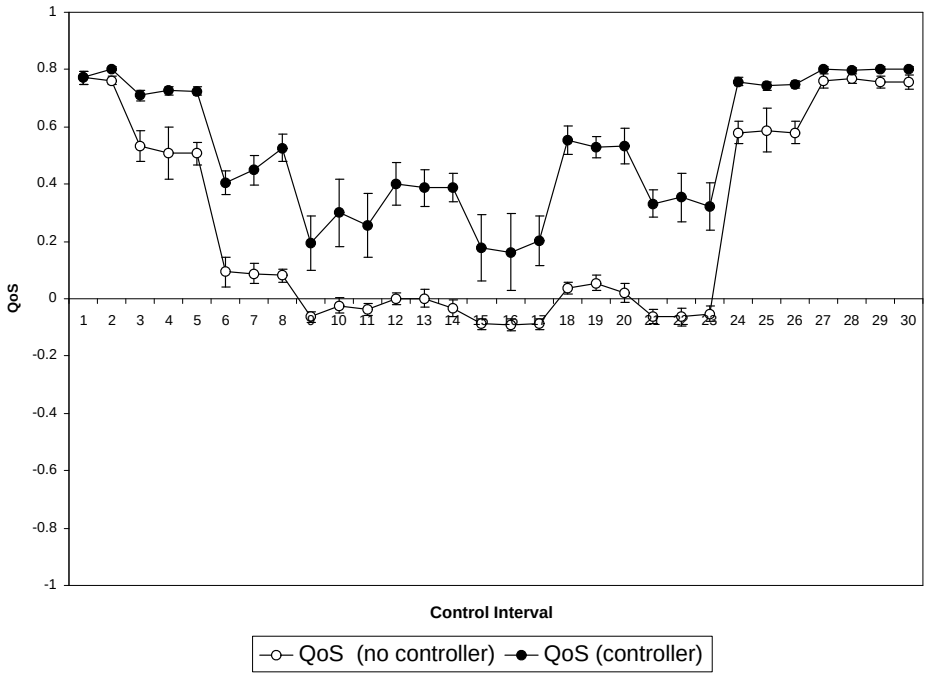


Figure 4.33: Variation of the global QoS function for the case of R-higher-weight.

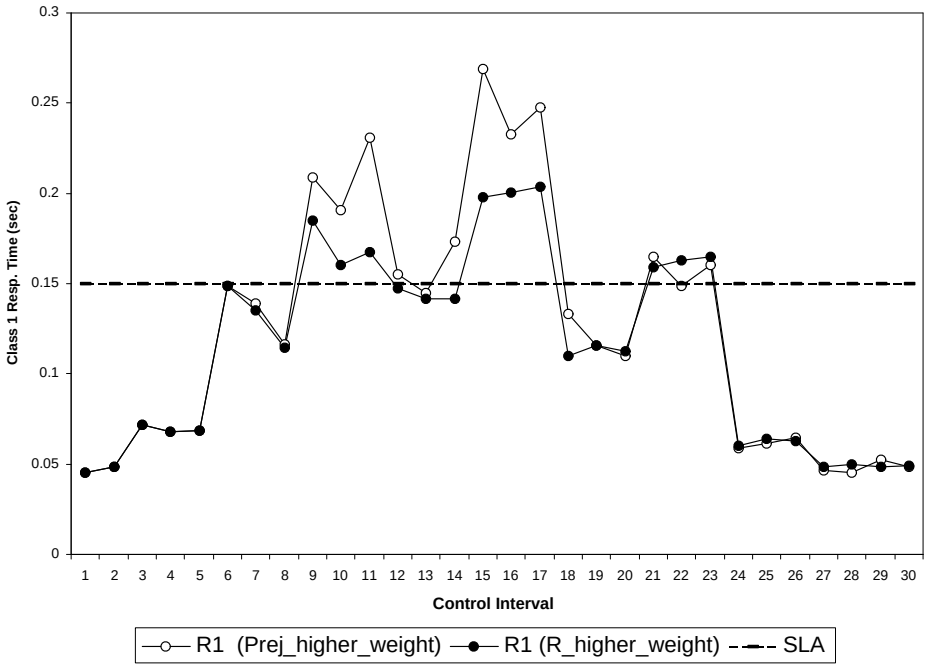


Figure 4.34: Variation of the global QoS function for the case of R-higher-weight.

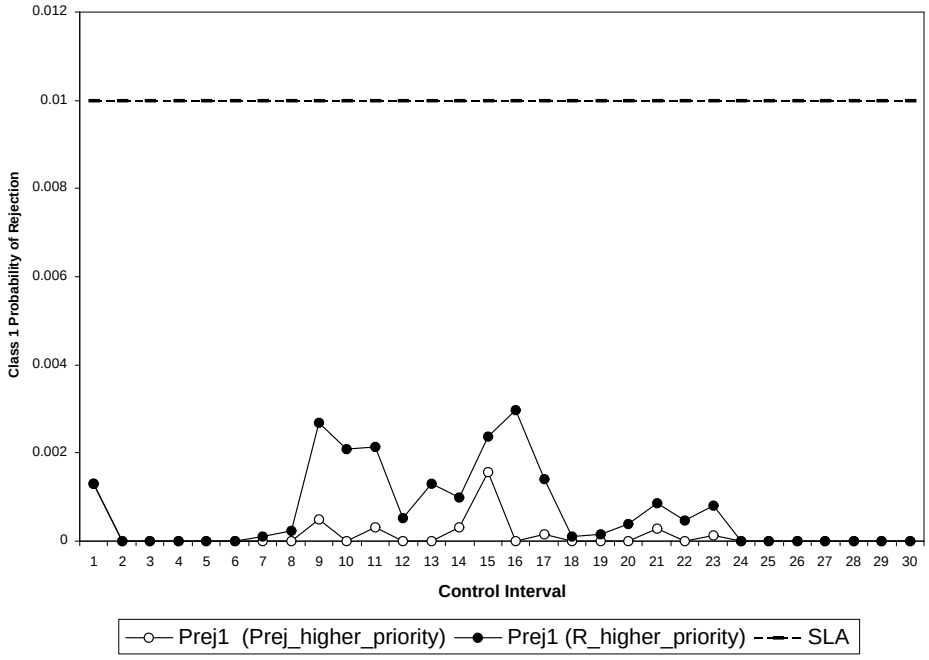


Figure 4.35: Variation of the global QoS function for the case of R-higher-weight.

Chapter 5: An Autonomic Data Center

This chapter shows how analytic performance models can be very beneficial to another category of autonomic systems. Hence, in the context of autonomic Internet data centers, we demonstrate the ability of our proposed controller approach in maintaining the centers operating at adequate performance levels. In this chapter, performance attributes are expressed in terms of utility functions as introduced in Chapter 2. The case of Internet data centers is chosen because of the growing interest of businesses into shifting the burden of operating and maintaining their IT infrastructures to third parties. As a result, hosting Internet data centers find themselves committed to provide good service levels to multiple customers using their limited IT resources. Autonomic computing helps then to ensure that resources are deployed dynamically in an intelligent manner, where they are mostly needed within the data center. It is a common practice to assign to each customer's application a logically self-contained environment, referred to as an Application Environment (AE).

The first section of this chapter provides a formal statement of the problem of dynamic resource allocation in Internet data centers. In the second section, an illustration of the control approach for the case of a simulated Internet data center is presented. In particular, that section provides a description of the proposed controller system, reviews the controller approach as well as the performance models for two types of hosted application environments: online transactions and batch. Simulation results

are presented in that section as well. The third section shows the efficiency of the controller approach for the case of a data center prototype. It starts with a description of the autonomic data center prototype, then presents the performance model used, and follows that by a discussion of experimental results. Section 4 presents a comparison to a machine learning approach, namely reinforcement learning, to achieve self-management for the prototype data center. It also contrasts this technique with the analytic performance models and discusses the pros and cons of each of these two approaches. Trade-offs between availability and performance in autonomic data centers is the subject of the last section where simulation results are presented as well. Work presented in this chapter appears in the following publications: [11, 55, 69]. The work on the data center prototype was conducted at IBM T. J. Watson Research Center while I was an intern in the summer of 2005.

5.1 The Dynamic Resource Allocation Problem

A data center consists of M application environments (AEs) and N servers (see Fig. 5.1). 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. In this work, we assume the presence of a Storage Area Network (SAN) in the data center so that the overhead induced by the switching of servers among AEs is always considered to be negligible.

AEs may run *online* transaction workloads or *batch* workloads. For the case of

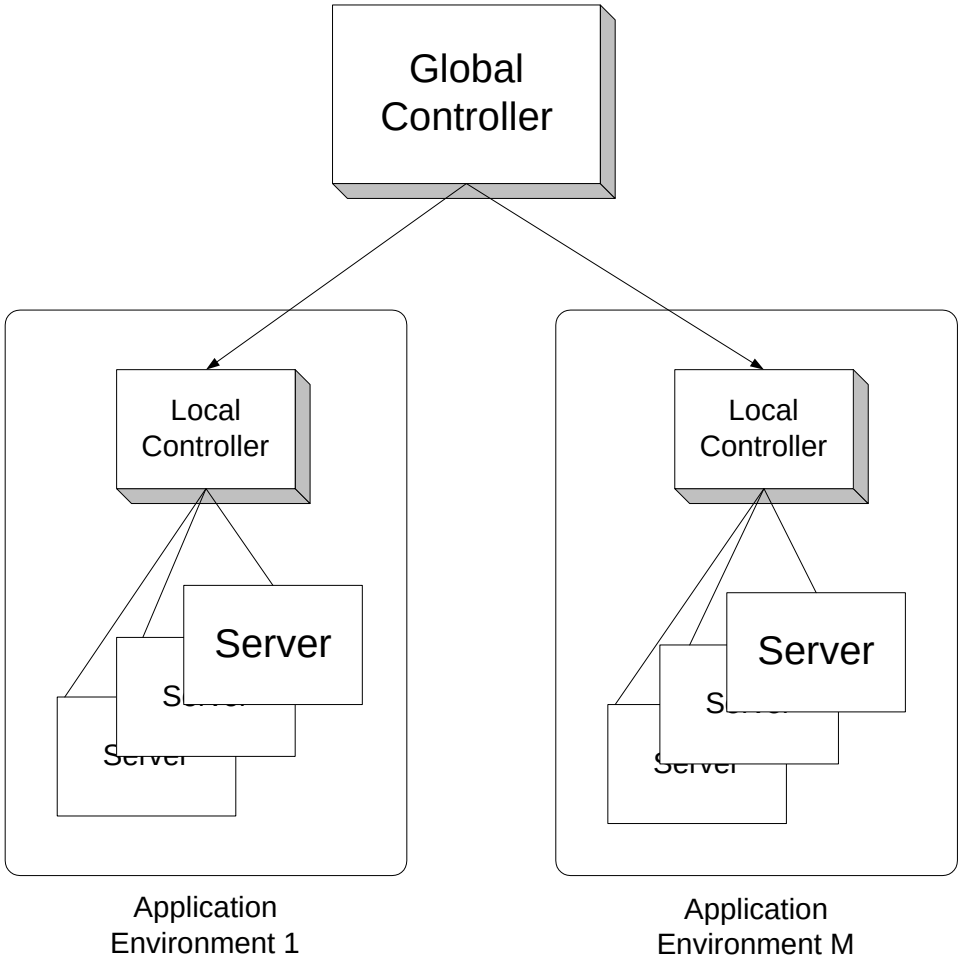


Figure 5.1: Application Environments.

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

$$\vec{w}_i = \begin{cases} (\lambda_{i,1}, \cdots, \lambda_{i,S_i}) & \text{if } AE_i \text{ is online} \\ (c_{i,1}, \cdots, c_{i,S_i}) & \text{if } AE_i \text{ is batch} \end{cases} \tag{5.1}$$

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 $\vec{R}_i = (R_{i,1}, \cdots, R_{i,S_i})$ and $\vec{X}_i = (X_{i,1}, \cdots, X_{i,S_i})$, 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,

$$\vec{R}_i = \mathcal{M}_i^o(\vec{w}_i, n_i) \tag{5.2}$$

$$\vec{X}_i = \mathcal{M}_i^b(\vec{w}_i, n_i) \tag{5.3}$$

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 5.2.3.1 and 5.2.3.2

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,

$$U_i = \begin{cases} f(\vec{R}_i) = f(\mathcal{M}_i^o(\vec{w}_i, n_i)) & \text{if } AE_i \text{ is online} \\ g(\vec{X}_i) = g(\mathcal{M}_i^b(\vec{w}_i, n_i)) & \text{if } AE_i \text{ is batch.} \end{cases} \tag{5.4}$$

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

$$U_g = h(U_1, \cdots, U_M). \tag{5.5}$$

We describe now the utility functions that we consider in this chapter. 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. Sigmoid functions possess such features and they have been deployed previously as

utility functions for autonomic data centers in [71]. Likewise, we as well use sigmoid-based utility functions for the online and batch AEs considered in this chapter. Thus, the utility function that we use for class s at online AE_i is defined as

$$U_{i,s} = \frac{K_{i,s}.e^{-R_{i,s}+\beta_{i,s}}}{1+e^{-R_{i,s}+\beta_{i,s}}} \tag{5.6}$$

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. 5.2 for $K_{i,s} = 100$ and $\beta_{i,s} = 4$.

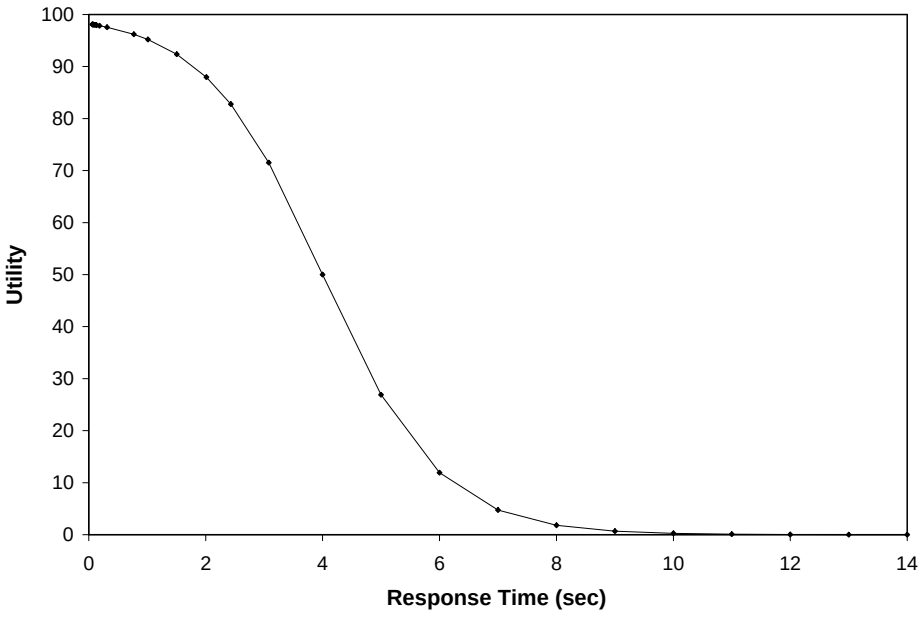


Figure 5.2: Utility as a function of response time.

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

$$U_i = \sum_{s=1}^{S_i} a_{i,s} \times U_{i,s} \tag{5.7}$$

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

$$U_{i,s} = K_{i,s} \times \left(\frac{1}{1 + e^{-X_{i,s}+\beta_{i,s}}} - \frac{1}{1 + e^{\beta_{i,s}}} \right) \tag{5.8}$$

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. (5.8) is defined in such a way that the utility is zero when the throughput is zero. See an example in Fig. 5.3 for $K_{i,s} = 100$ and $\beta_{i,s} = 2$.

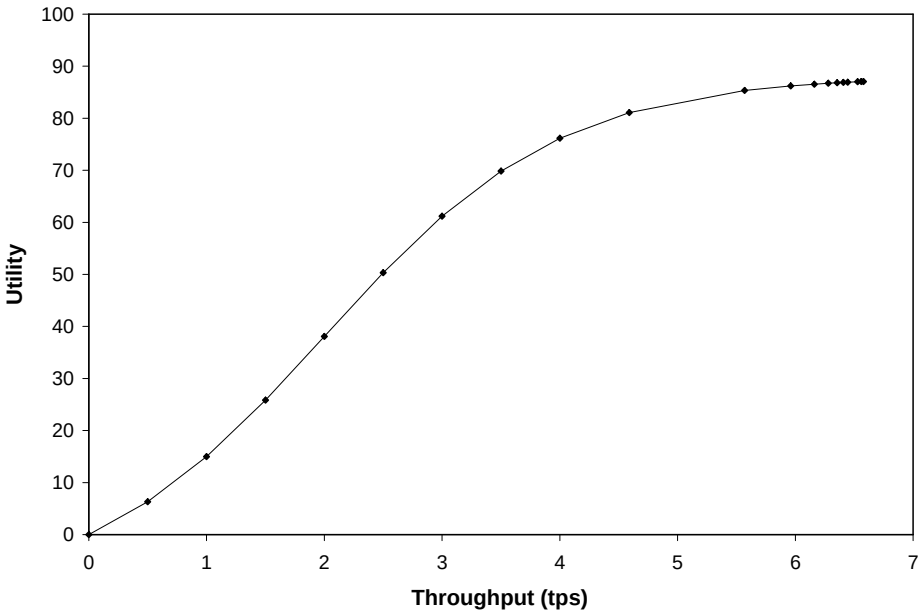


Figure 5.3: Utility as a function of the throughput.

We use Eq. (5.7) to compute U_i in the batch case also. The resource allocation problem addressed in this chapter is similar to the one addressed in [71]. However, our solution uses a different approach, which is based on the concepts we developed

in our earlier work [10, 46, 51, 48]. 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.

5.2 Illustration of the Control Approach on a Simulated Internet Data Center

In this section we examine our proposed control approach for the case of a simulated data center. We start by providing a description of the simulated system. We then revise the controller algorithm, present the performance models used, and provide simulations results.

5.2.1 System Description

The simulated Internet data center consists of a set of AEs. Some AEs are of type online transactions while the others are of type batch processing. Each AE has one or more classes of customer requests. The classes of an online transaction AE are all open. For a batch processing AE, all the classes are closed. Each AE has its own SLA per class, in terms of maximum response times and minimum throughputs for the case of online transaction AEs and batch AEs, respectively. AEs local controllers keep track of the workloads of all classes of the AE as well as the assigned number of resources (servers in this case). Local controllers communicate these information to a global controller at its request. The global controller then runs a controller algorithm to determine the optimal number of resources to be assigned to each AE. Local controllers are then instructed of the changes in their allocated resources. In this sense, the configurable knobs are simply the number of servers to assign to each

AE.

5.2.2 The Controller Approach

Figure 5.4 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 [3] 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 5.1. 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 5.5 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 $\vec{n} = (n_1, \cdots, n_i, \cdots, n_M)$. For each configuration examined by the controller algorithm, the corresponding number of servers (3) is sent

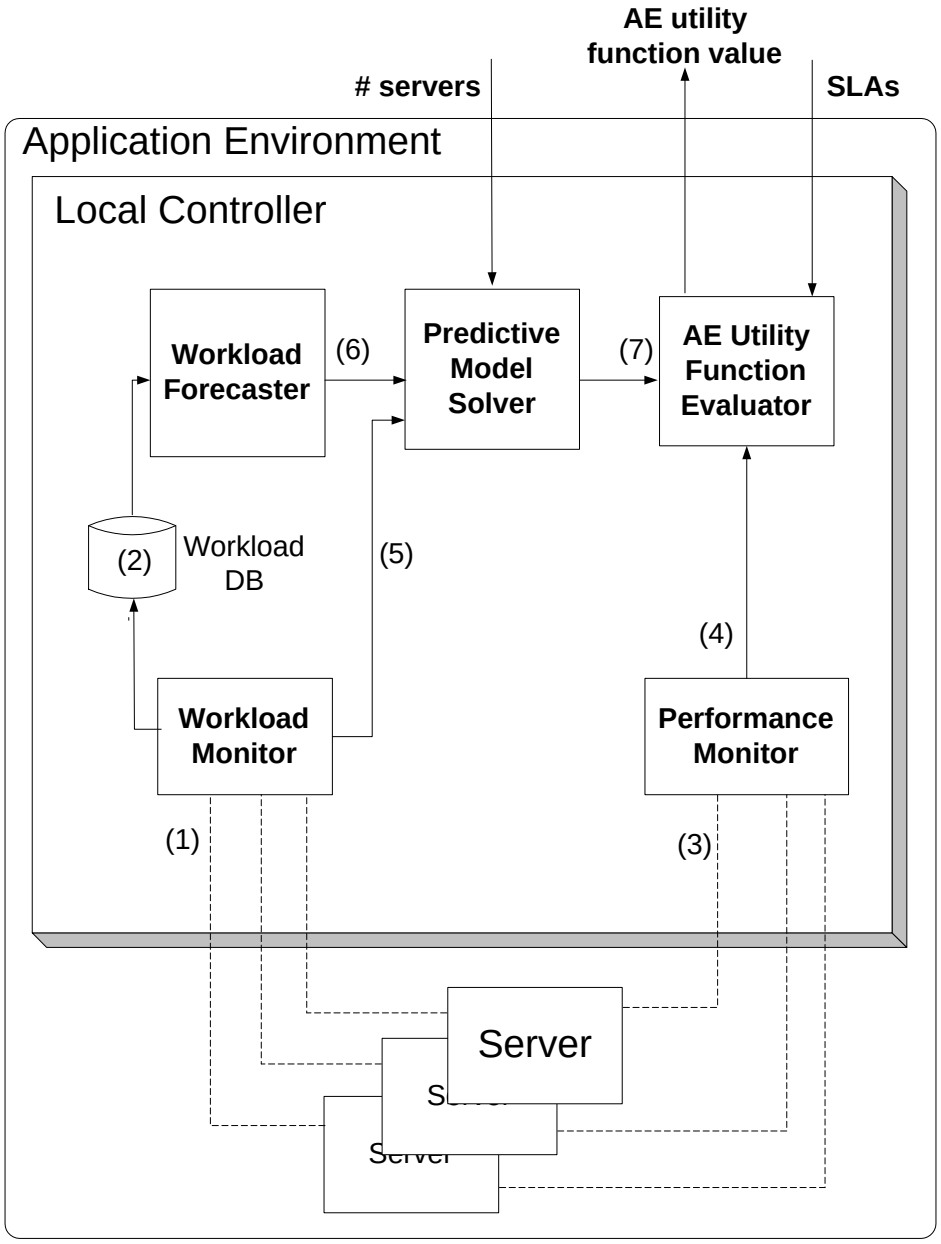


Figure 5.4: Local Controller.

to each AE, which return the values of the utility functions U_i (4). The *global utility function evaluator* computes U_g using Eq. 5.5 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 $\vec{n} = (n_1, \cdots, n_M)$ using a beam-search algorithm. Beam search is a combinatorial search procedure that works as follows [63]. Starting from the initial configuration, the global utility function, U_g, is computed for all 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 $\vec{v} = (v_1, v_2, \cdots, v_M)$ be a *neighbor* of a current configuration vector $\vec{n} = (n_1, n_2, \cdots, n_M)$. 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).

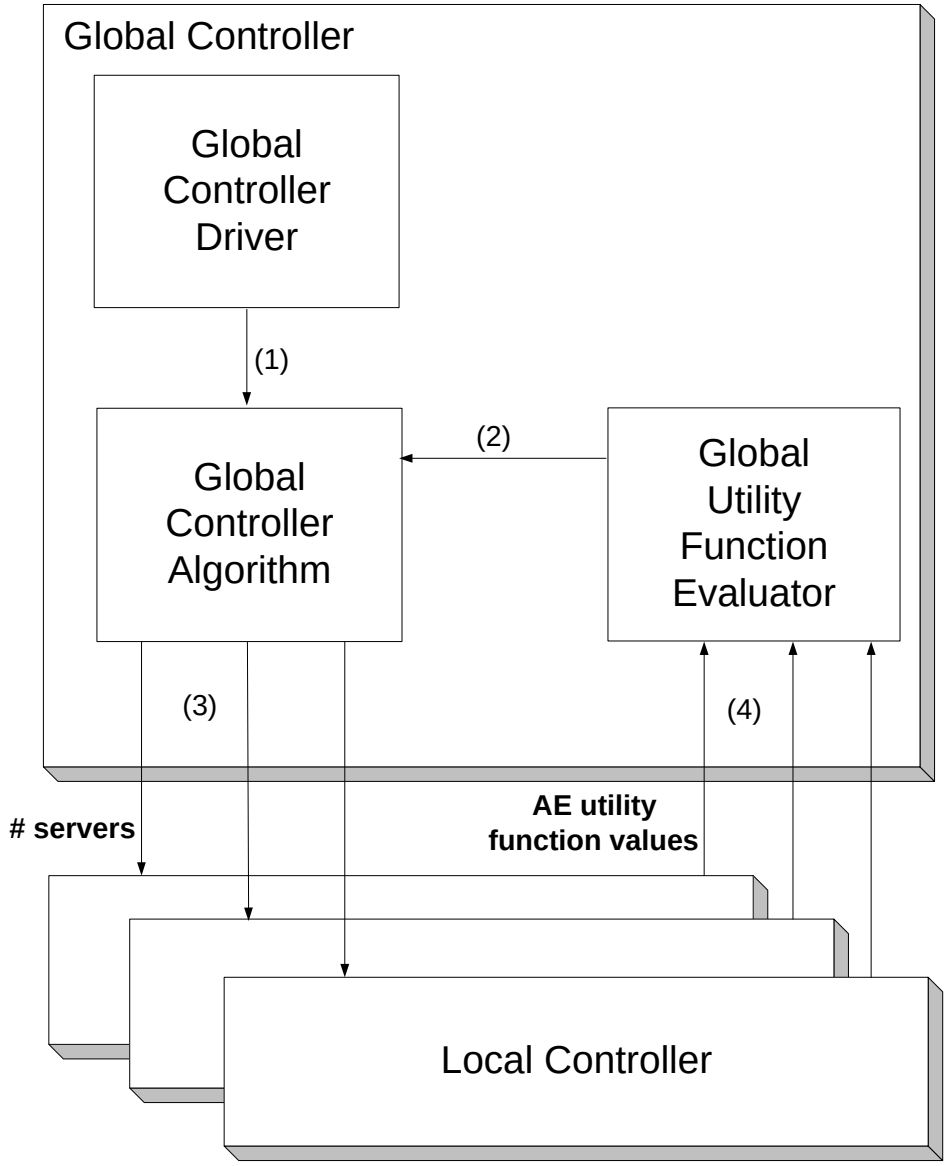


Figure 5.5: Global Controller.

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 successor 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

1. $\sum_{i=1}^{M} v_i = N$,
2. The utilization of any resource within the online servers of online AEs does not exceed 100%, and
3. 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 Fig. 5.6. The following notation is in order:

- $\mathcal{V}(\vec{n})$: set of neighbors of configuration vector $\vec{n}$.
- LevelList$_i$: set of configuration vectors examined at level i of the beam search tree.
- CandidateList: set of all configuration vectors selected as the k best at all levels of the beam search tree.
- 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.

```
LevelList_0 ← n⃗_0;
CandidateList ← LevelList_0;
For i = 1 to d Do
    Begin
       LevelList_i ← ∅;
       For each n⃗ ∈ LevelList_{i−1} Do
           LevelList_i ← LevelList_i ∪ V(n⃗);
       LevelList_i ← Top (k, LevelList_i);
       CandidateList ← CandidateList ∪LevelList_i;
    End;
n⃗_opt ← max (CandidateList)
```

Figure 5.6: Controller Algorithm.

5.2.3 The Performance Models

The analytic performance models that are used by the controller to predict the global utility, U_g, for a given configuration vector $\vec{n} = (n_1, \cdots, n_M)$ are presented in this section. We distinguish between the performance models used for each type of application environment. The reason for that comes from the fact that the classes for customer requests are all open for online AEs and closed for batch AEs. It is important to mention that both performance models scale very well with respect to the number of classes in each AE.

5.2.3.1 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 $D_{i,s}^{\text{CPU}}$ 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 $D_{i,s}^{\text{IO}}$ 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 [42], 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 [42], the response time, $R_{i,s}(n_i)$, of class s transactions at AE_i can be computed as:

$$R_{i,s}(n_i) = \frac{D_{i,s}^{CPU}}{1 - \sum_{t=1}^{S_i} \frac{\lambda_{i,t}}{n_i} \times D_{i,t}^{CPU}} + \frac{D_{i,s}^{IO}}{1 - \sum_{t=1}^{S_i} \frac{\lambda_{i,t}}{n_i} \times D_{i,t}^{IO}}. \quad (5.9)$$

Note that the response time $R_{i,r}$ is a function of the number of servers, n_i, allocated to AE_i. Eq. (5.9) assumes perfect load balancing among the servers of an AE. Relaxing this assumption is straightforward.

Figure 5.7 shows response time curves obtained with Eq. (5.9) 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 5.1.

5.2.3.2 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 these cases, throughput is often of higher concern than response time. Multiclass closed

Table 5.1: Service demands (in sec) for the example of Fig. 5.7

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

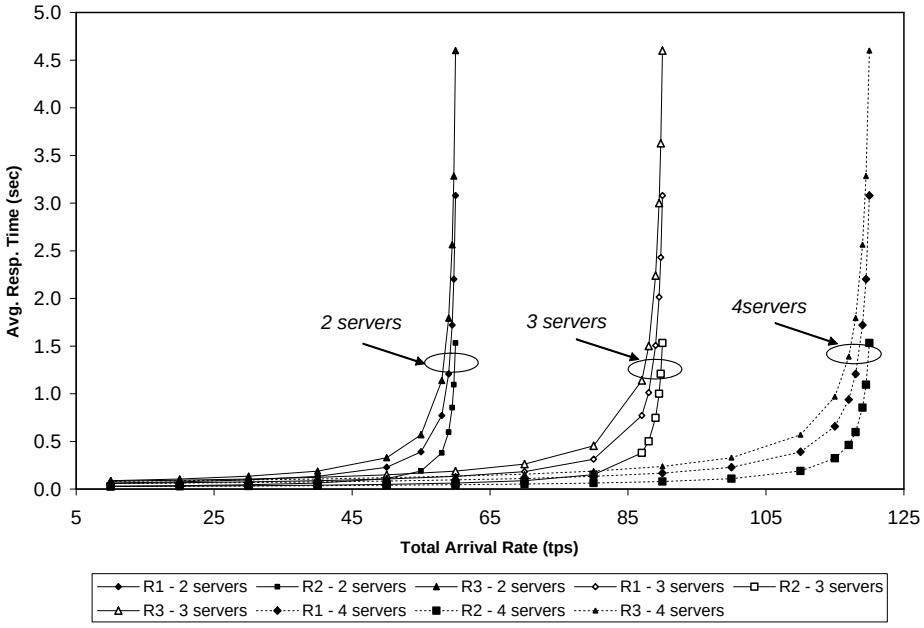


Figure 5.7: Response times for an Online AE.

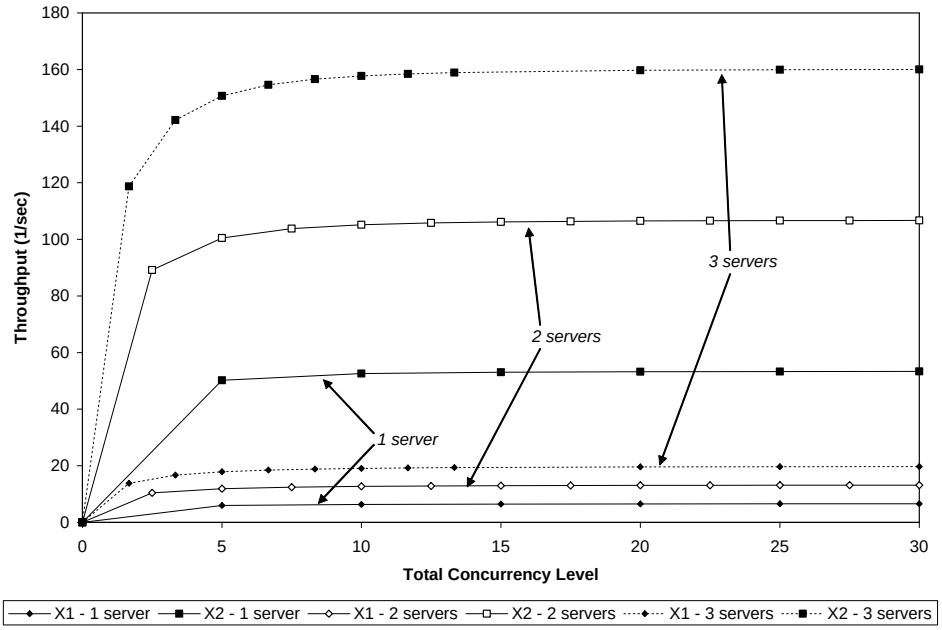


Figure 5.8: Throughputs for a batch processing AE.

queuing network models [42] 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 5.8 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 [42], 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 5.1.

5.2.4 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 [16] 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.

Table 5.2 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),

Table 5.2: Input Parameters for the Experiments

Application Environment 1			
Type	Online		
S_1	3		
s	1	2	3
$D_{1,s}^{\text{CPU}}$	0.030	0.015	0.045
$D_{1,s}^{\text{IO}}$	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	
$D_{2,s}^{\text{CPU}}$	0.030	0.015	
$D_{2,s}^{\text{IO}}$	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
$D_{3,s}^{\text{CPU}}$	0.005	0.010	0.015
$D_{3,s}^{\text{IO}}$	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

and the weights ($a_{i,s}$ values) used to compute the utility function U_i according to Eq. (5.7). The values of β for the two online AEs are given in seconds because they correspond to response times. The β 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

$$U_g = 0.35 \times U_1 + 0.35 \times U_2 + 0.3 \times U_3. \tag{5.10}$$

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,

$$K_{i,s} = 100 \left(\frac{1 + e^{\beta_{i,s}}}{e^{\beta_{i,s}}} \right) \tag{5.11}$$

for the two online AEs. For the batch AE, the scaling factor for each class was computed in a way that its value is 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 [42], 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,

$$X^+_{i,s} = \frac{N - (M-1)}{\max(D^{\text{CPU}}_{i,s}, D^{\text{IO}}_{i,s})} \tag{5.12}$$

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

$$K_{i,s} = \frac{100}{\left(\frac{1}{1+e^{-X^+_{i,s}+\beta_{i,s}}} - \frac{1}{1+e^{\beta_{i,s}}} \right)}. \tag{5.13}$$

.

During the experiments, the transaction arrival rates at the two online AEs was varied as shown in Fig. 5.9. 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,

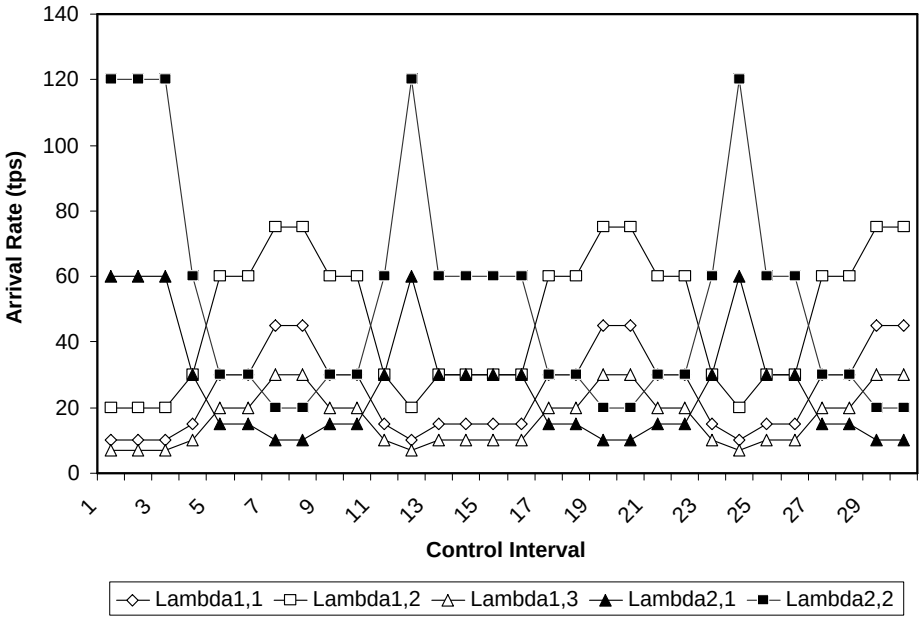


Figure 5.9: Variation of the workload intensity for AEs 1 and 2.

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.

5.2.5 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 these values and allowed the controller to obtain the best allocation of servers to AEs. The resulting initial configuration thus obtained was $n_1 = 7$, $n_2 = 8$,

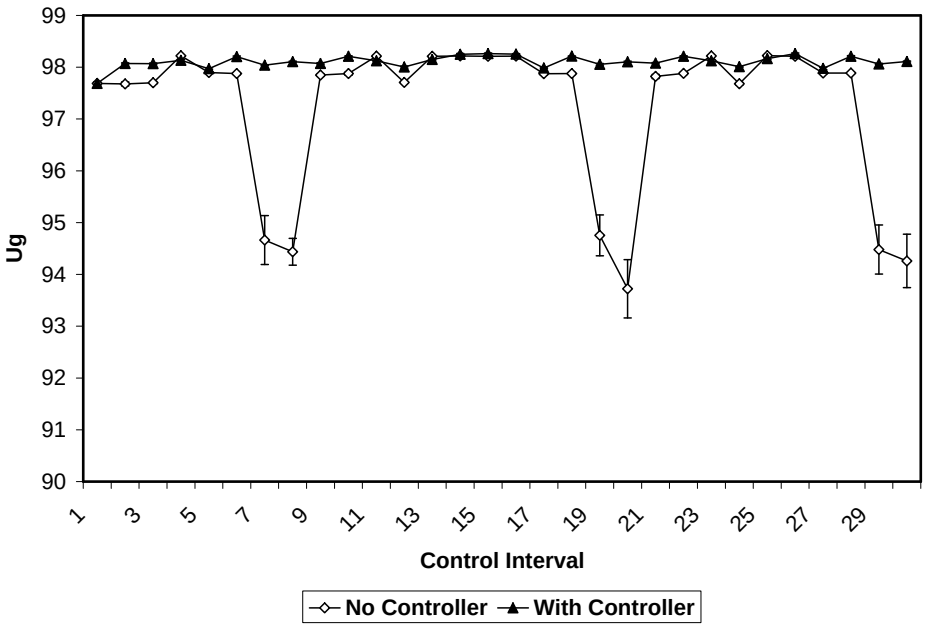


Figure 5.10: Variation of the global utility function U_g.

and $n_3 = 10$.

Figure 5.10 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 mantains 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. 5.9. 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. 5.11. The figure shows that U_1 and U_2 decrease when the corresponding arrival rates reach their peak. The figure also

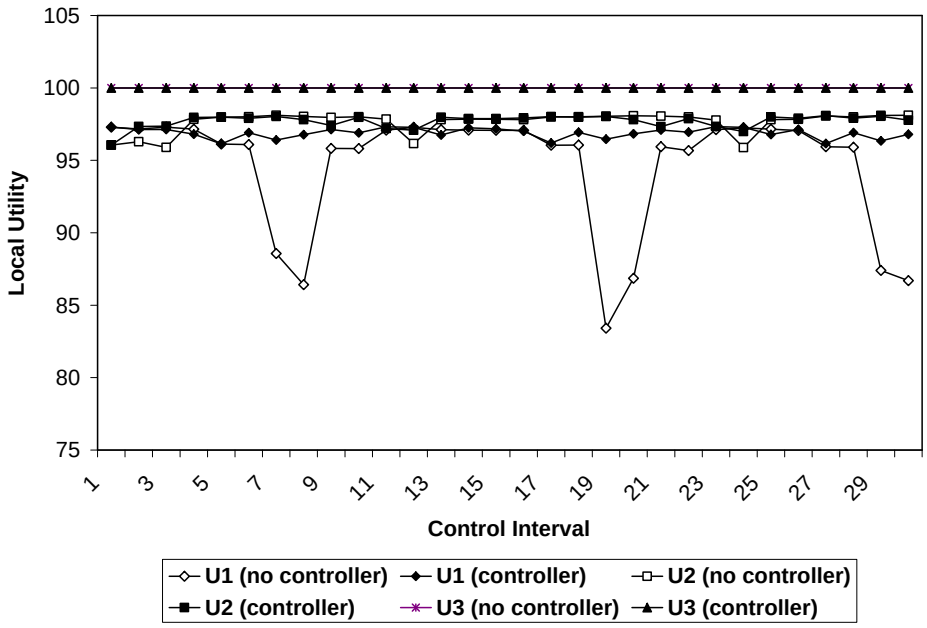


Figure 5.11: Variation of the local utility functions U_1, U_2, and U_3.

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 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 5.12 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 initially, unlike AE_1. At CI = 4, the arrival rates for AE_2 start to increase and the arrival rates for AE_1 start to decrease (see Fig. 5.9). 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 happen for AEs 1 and 2. This observation is explained by the fact that 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 5.13 shows the variation 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.

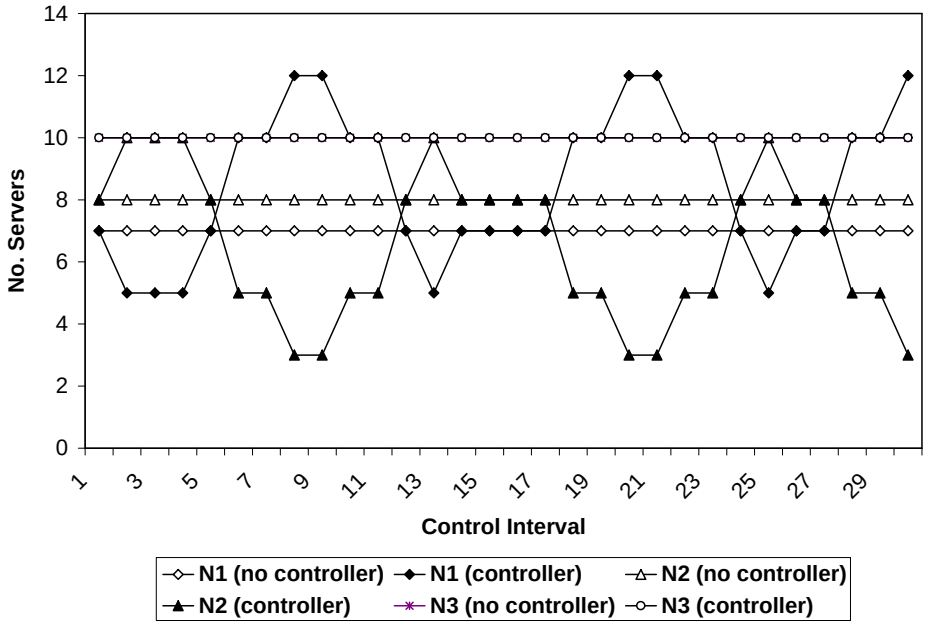


Figure 5.12: Variation of the number of servers n_1, n_2, and n_3.

A similar analysis for AE_2 is shown in Fig. 5.14 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 5.15 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

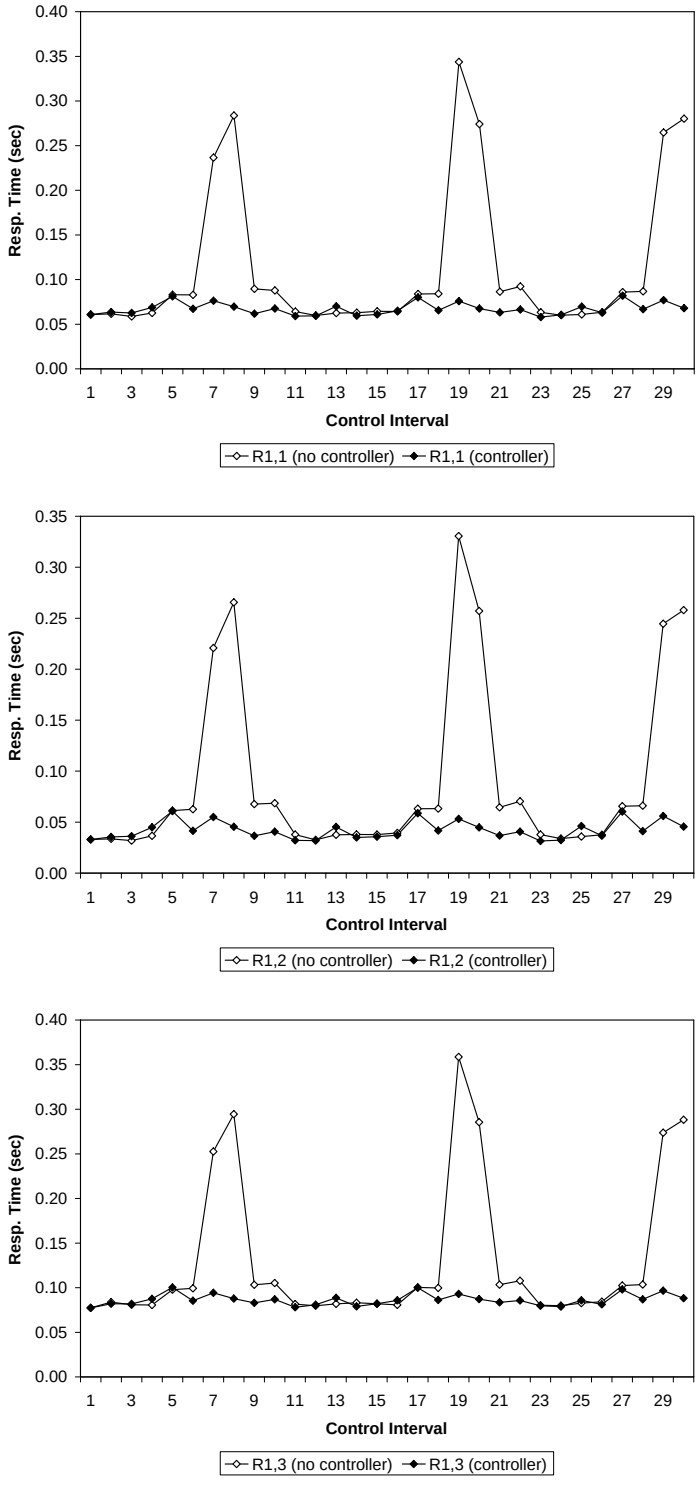


Figure 5.13: Variation of the average response times $R_{1,1}$ (top), $R_{1,2}$ (middle), and $R_{1,3}$ (bottom).

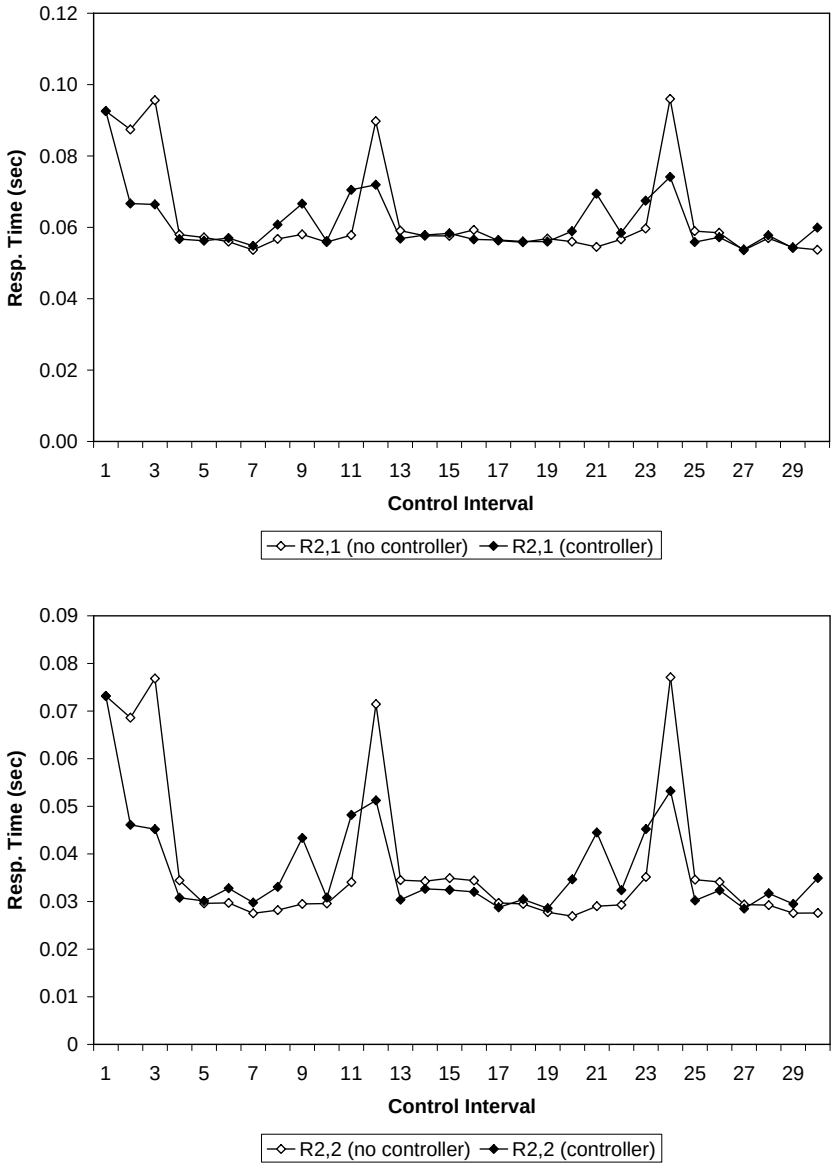


Figure 5.14: Variation of the average response times $R_{2,1}$ (top) and $R_{2,2}$ (bottom) for AE_2.

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 5.16 shows a situation similar to that of Fig. 5.15, 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 discussed before, we ran an additional experiment to force servers to be given to the batch AE, AE_3. For that purpose, we maintained the same input parameters for AEs 1 and 2 and made the following changes to the parameters of AE_3: $D_{3,1}^{\text{CPU}} = 0.004$ sec, $D_{3,2}^{\text{CPU}} = 0.006$ sec, $D_{3,3}^{\text{CPU}} = 0.008$ sec, $D_{3,1}^{\text{IO}} = 0.002$ sec, $D_{3,2}^{\text{IO}} = 0.004$ sec, $D_{3,3}^{\text{IO}} = 0.006$ 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 5.2. The new SLAs for AE_3 are lower, and the concurrency levels for each class 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. 5.17 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

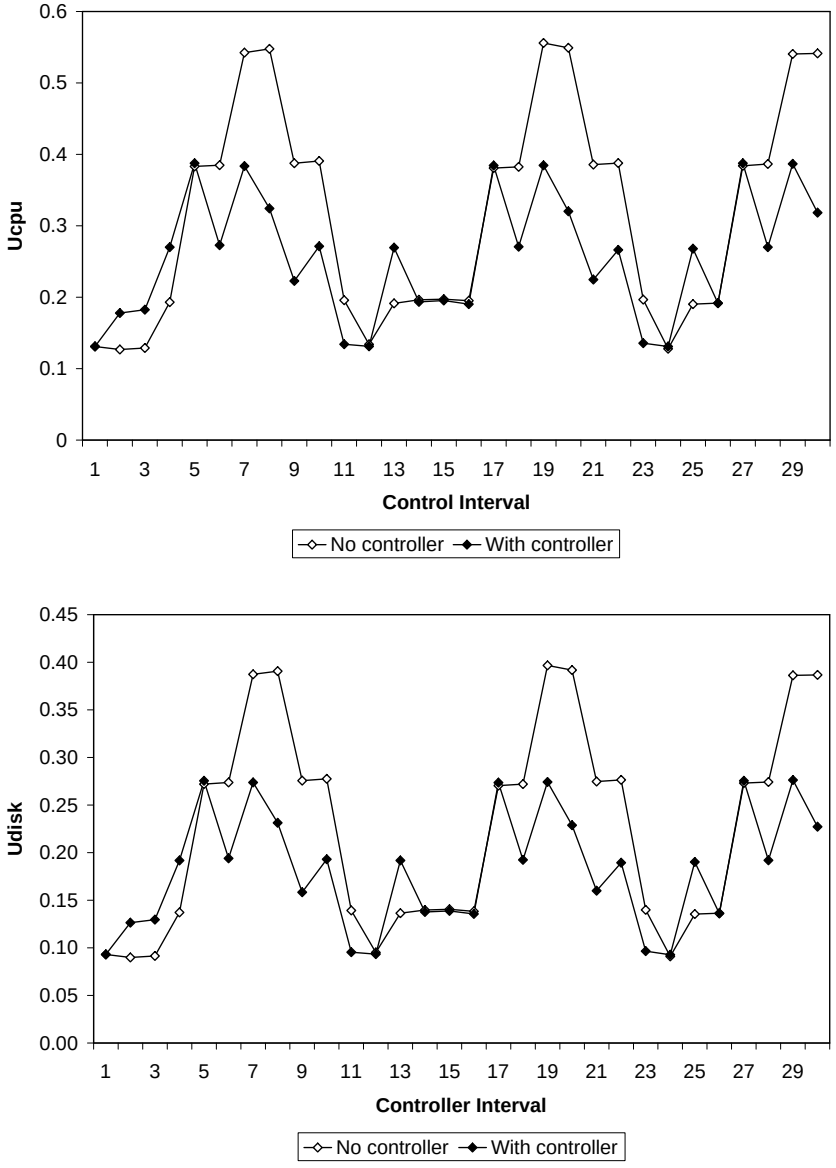


Figure 5.15: Utilization of the CPU and disk for AE_1.

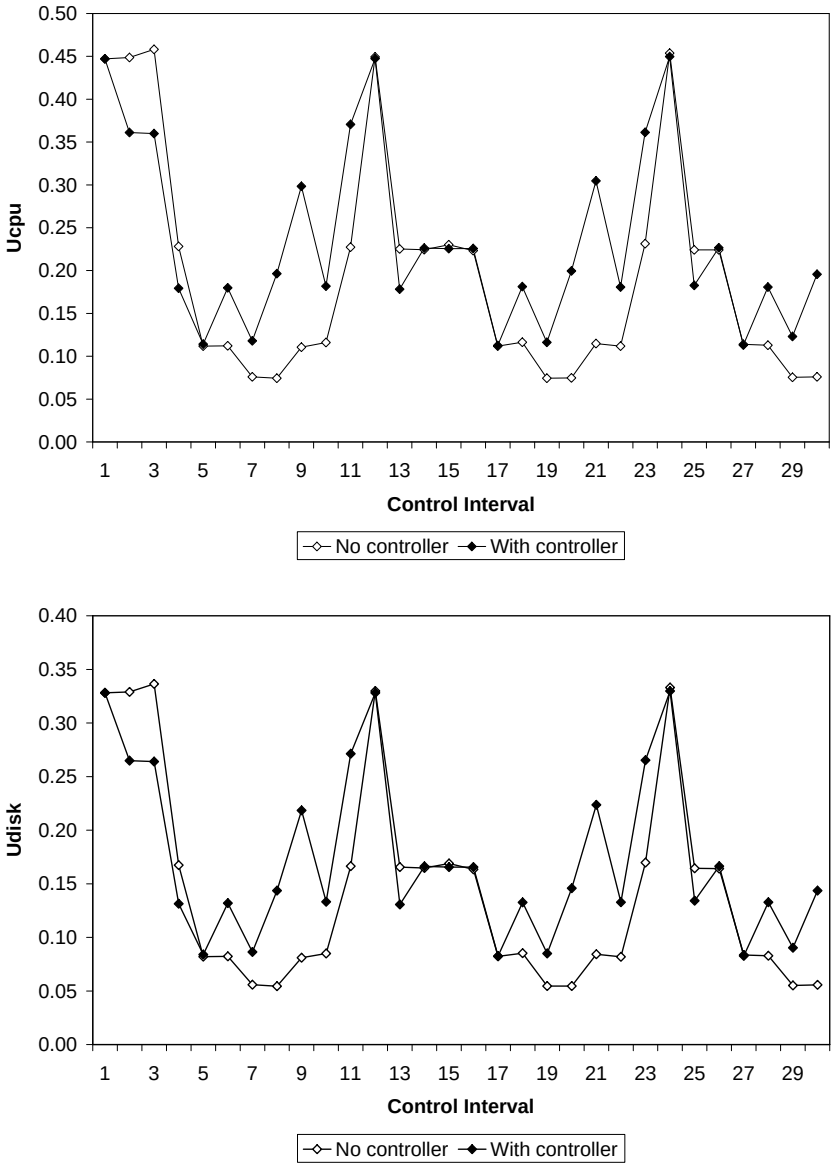


Figure 5.16: Utilization of the CPU and disk for AE_2.

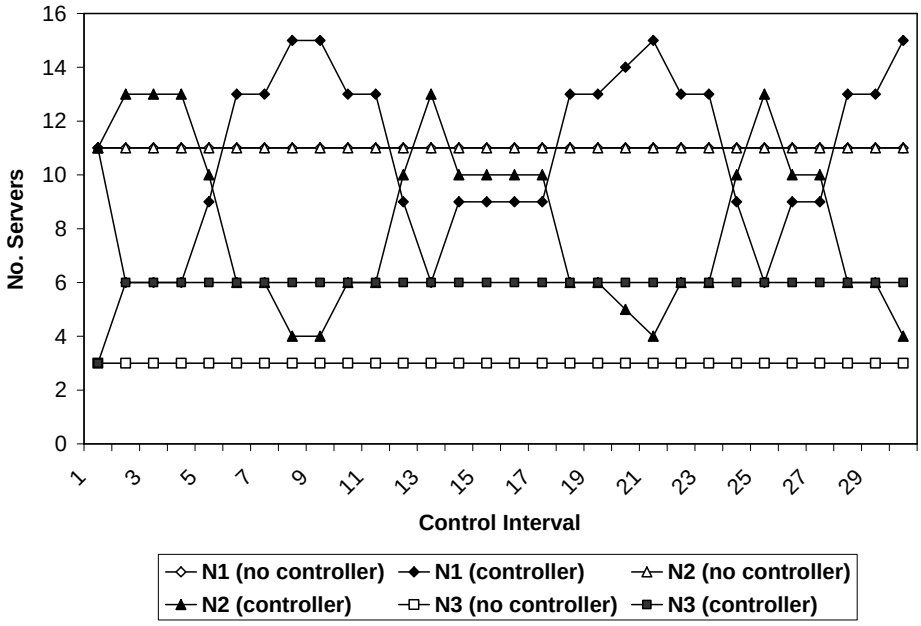


Figure 5.17: Variation of the number of servers n_1, n_2, and n_3 during the experiments with an initial allocation of 3 servers to AE_3.

alternate between AE_1 and AE_2 with n_1 and n_2 ranging between 6 and 15 and between 4 and 13, respectively.

5.3 Illustration of the Control Approach on an Internet Data Center Prototype

In this section we examine the effectiveness of analytic performance models for the case of a real Internet data center prototype, called Unity, availabale at IBM T. J. Watson Research Center. We start by providing a description of the system under test. Then, we discuss the controller algorithm, present the performance model used, and provide experimental results later. Work reported in this section and the following

one was conducted at IBM T. J. Watson Research Center in the summer of 2005.

5.3.1 System Description

The Internet data center prototype consists of a finite number of homogeneous servers and it hosts a set of AEs. Each server in the data center runs an application server (WebSphere Basic 5.0) on top of a database management system (DB2). Servers are assigned to AEs in an exclusive manner at any time. Therefore, during a given controller interval, servers service requests for one and only one AE. Some AEs are of type online transactions while others are of type batch processing. Each AE has a single class of customer requests. The batch application is considered to have a constant demand therefore its SLA is simply an increasing function of the number of assigned servers. Online transaction AEs run the Trade3 application [70]. Trade3 is a realistic simulation of an electronic trading platform that is designed to benchmark web servers. At any time, there is a finite number of customers that alternate between sending requests to the the online transaction AEs, waiting for responses, thinking for a certain amount of time, sending the next requests and so and so forth. The SLAs for online transaction AEs are expressed in terms of desired maximum response times. A sigmoidal function is used to map achieved service levels (S) to utiliy values (U). This function is also known as a service level utility function, $U(S)$. However, since a service level, S, is a function of the resource level (i.e., number of assigned servers), n, the utility values can be expressed in terms of n directly. In that case, $U(n)$ is referred to as a resource level utility function.

Each AE has its own Application Manager while there is a single Resource Arbiter

in the data center. The Resource Arbiter acts like a global controller whereas Application Managers act like local controllers. Application Managers are in charge of keeping track of their workloads, their number of assigned servers, n, and of running an optimization algorithm to determine a resource level utility function, $U(n)$, that maps resource levels to utility values. Application Managers need to send their resource level utility function to the Resource Arbiter which in turn runs a simple algorithm to find the allocation that maximizes the sum of all resource level utility functions. Therefore, the global utility of the data center, U_g, is defined as: $U_g = U_{transaction} + U_{batch}$.

The Resource Arbiter and the Application Managers are invoked every 5 seconds. The system is illustrated in Fig. 5.18 and is found in [71].

Each online transaction AE is equipped with a dispatcher that assigns incoming requests to servers within the AE in a round robin fashion so as to maintain an equally balanced load across all servers.

5.3.2 The Controller Approach

The control in the data center prototype is achieved through a coordination between the Resource Arbiter and the Application Managers as shown in Fig. 5.18. The Resource Arbiter is a quite light optimization algorithm that does not need to know about the details of the associations between assigned resource levels to AEs and the resulting service levels. On the other hand, Application Managers are responsible of the mapping between resource levels and the expected service levels. This mapping results in a curve of service levels, S, as a function of resource levels, n. This curve is denoted by $S(n)$. The curve $S(n)$ is obtained by running some analytic performance

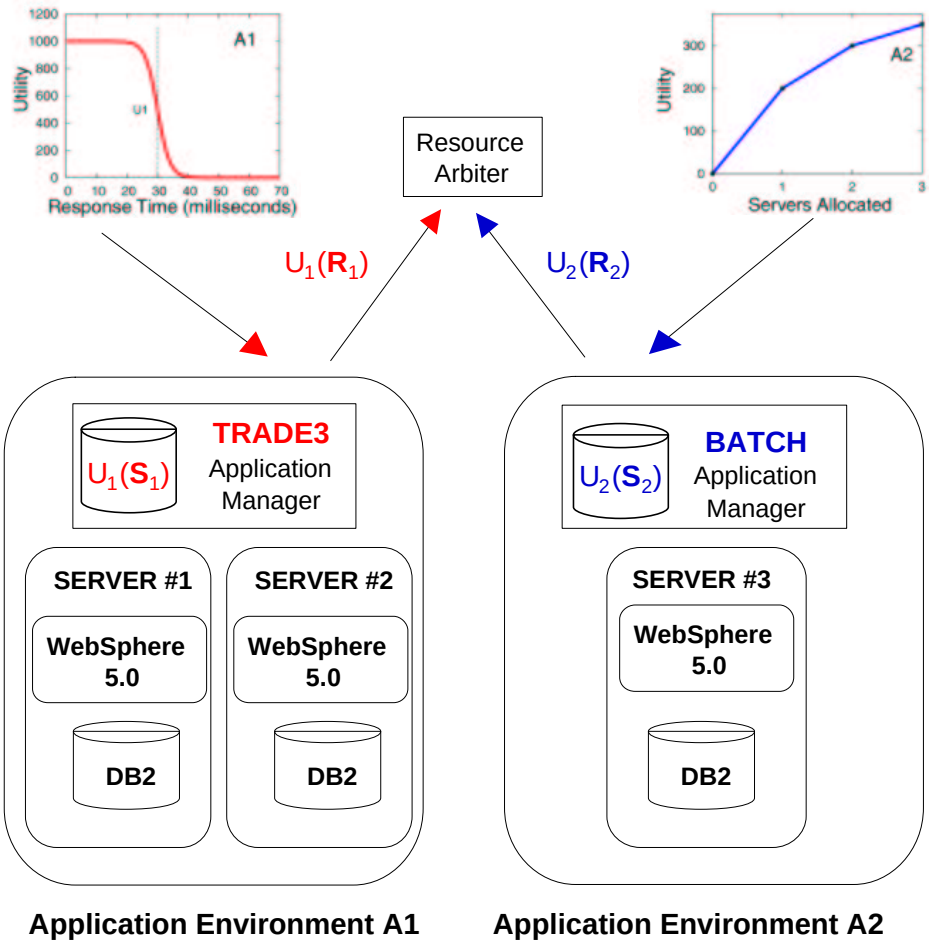


Figure 5.18: IBM Data Center Prototype.

models. Once the Application Managers have computed their $S(n)$ curve, they can map that back to a resource level utility curve, $U(n)$, through their maintained service level utility curve, $U(S)$. In other words, $U(n)$ is computed as $U(n) = U(S(n))$. At the start of each control interval, Application Managers send their computed resource level utility function, $U(n)$, to the Resource Arbiter. The Resource Arbiter determines then the best allocation of servers for each AE and then instructs the Application Managers for the new configuration deployment. The only constraint in the assignments of servers is that a transaction AE must have at least one server at any time while a batch AE might have none.

5.3.3 The Performance Model

As explained in section 5.3.1, the Application Managers for batch AEs simply used a linearly increasing function for $S(n)$. So the only performance model needed is that of an online transaction AE. A transaction AE has only a finite number of customers (M) and of servers (n) at any time. These customers alternate between sending HTTP requests to the transaction AE, waiting for responses, thinking for a certain amount of time (Z), and then sending their next requests and so and so forth. The average think time (Z) is varying as well. A system level model for a transaction application environment is shown in Fig. 5.19.

The performance model is used to predict the average response time, R, of the transaction application environment for each possible assignment level of servers as a function of input parameters M and Z. In other words, the performance model needs to evaluate $R_{t+1}(M_{t+1}, Z_{t+1}, n_{t+1})$ for all possible server assignment levels, n_{t+1}, for the controller interval $t+1$. At the end of the current controller interval t, the following

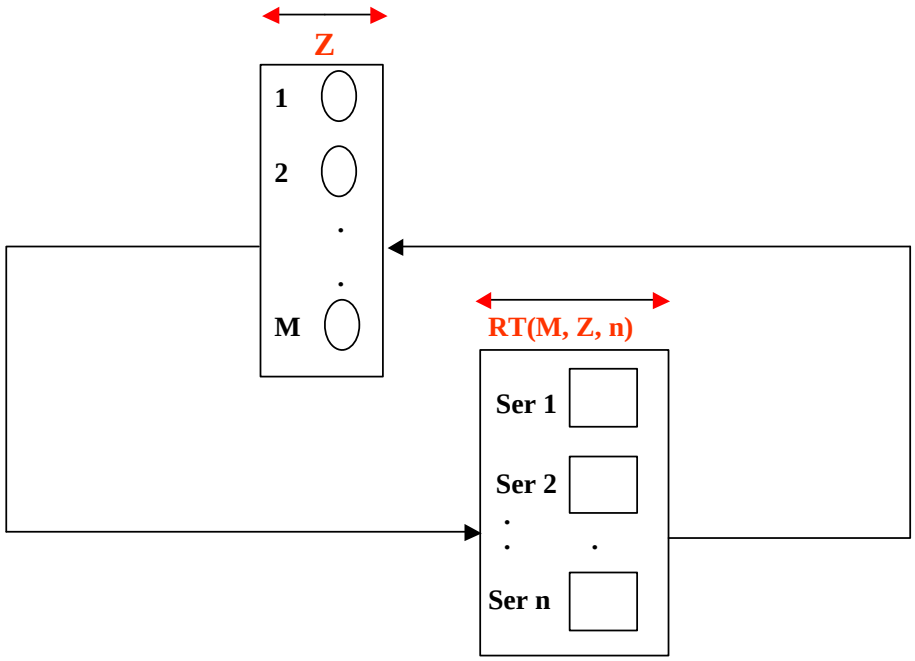


Figure 5.19: System Model for a Transaction AE.

metrics are known to the performance model: R_t, M_t, n_t, and X_{0_t} (the measured throughput of the entire application environment). Since the workload is equally distributed across the servers in a transaction AE, solving for $R_{t+1}(M_{t+1}, Z_{t+1}, n_{t+1})$ is equivalent to solving for $R_{t+1}(M_{t+1}/n_{t+1}, Z_{t+1}, 1)$. That corresponds exactly to the case of a transaction application environment with only one server and M_{t+1}/n_{t+1} customers each having an average think time equal to Z as shown in Fig. 5.20.

The well known Mean Value Analysis (MVA) technique is very suitable for solving such queuing models [42]. However, MVA needs the following input parameters: the average service time at the server, ST, and the average think time, Z_{t+1}. The performance model uses Z_t as an estimation for Z_{t+1}. Using an approximation from the M/M/1 model, ST can be estimated as $ST = \frac{R_t}{1+\lambda_t * R_t}$. However, this approximation

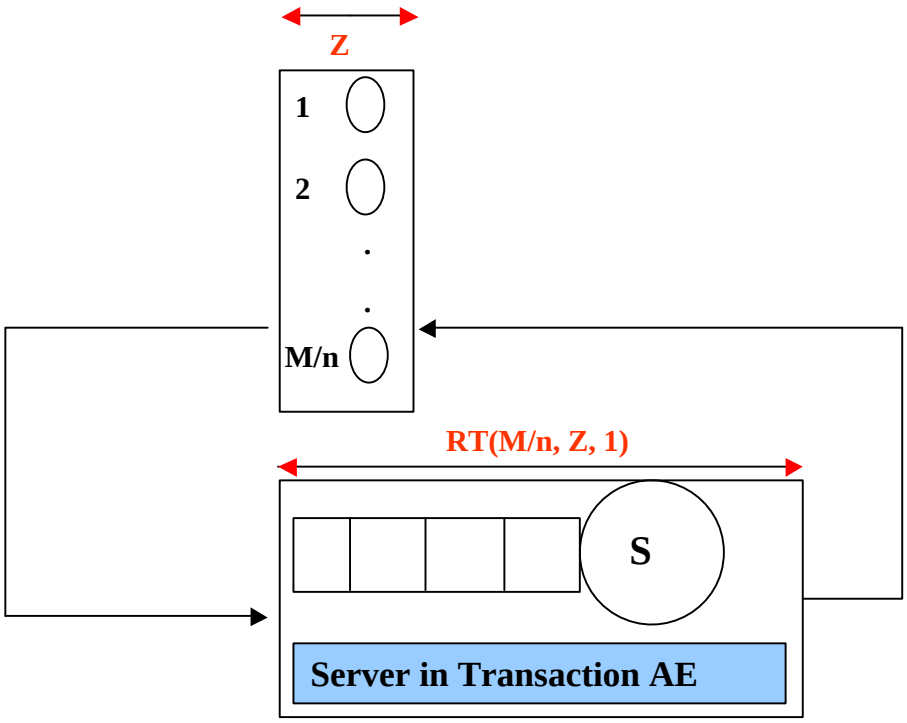


Figure 5.20: Queuing Model for a server in a Transaction AE.

is quite sensitive to the variations in R_t which can be mainly attributed to the overhead introduced by Java Garbage Collection process. Therefore, some exponential smoothing for ST is done before it is actually passed to the MVA algorithm. For the MVA second input parameter, Z_t, it is derived from the Interactive Response Time Law [42], as: $Z_t = \frac{M_t}{X_{0_t}} - R_t$.

It is worth mentioning that this suggested performance model is quite high level in the sense that it models the entire server as a single resource. Therefore, this performance model is oblivious to the interactions between the WebSphere and DB2 processes running in the server, to the dynamic behavior of the WebSphere pool of threads, and to the service demands at the CPUs and disks. The reason this performance model is chosen for control is to be able to conduct a fair comparison to another controller

that uses a black box machine learning approach as explained in section 5.3.4.

5.3.4 The Experimental Setting

The Unity data center prototype used for these experiments consists of a cluster of five identical IBM eServer xSeries 335 machines running Redhat Enterprise Linux Advanced Server. The data center hosts one online transaction AE and one batch AE. The online transaction AE runs the Trade3 benchmark. The SLA for the online transaction AE requires that the maximum average response time does not exceed 40 milliseconds. The resource level utility vector for the batch AE is (-20, 8, 34, 50, 80) for an allocation of 0, 1, 2, 3, or 4 servers, respectively. The batch AE cannot be granted the total number of servers in the data center (5 in this case) because the online transaction AE needs at least one server all the time.

For the transaction AE, the utility as a function of the measured response time, R, is computed as follows:

$$U_{transaction}(R) = \frac{200}{1 + e^{\frac{R-40}{5}}} - 100 \tag{5.14}$$

We conducted two sets of experiments. In the first set of experiments, the number of customers for the transaction AE is kept fixed at 45 and the think time is an exponentially distributed random variable. In the second set of experiments, the number of customers varies between 10 and 60 while the think time is drawn from an exponential distribution with a fixed mean equal to 167 msec. In either case, the workload varies according to a modified time series model of Web traffic that was developed by Squillante et al. [66]. In this modified time series, either the average think time or the average number of customers is reset every 1 second. The resulting

workload mimics fairly well the stochastic burstiness in demand typically found in Internet traffic. In each set of experiments, we compare the mean global utilitiy value of the data center under the four following server allocation techniques: random, static (3 servers for the online transaction AE and 2 servers for the batch AE), our suggested performance model, and a machine learning approach. For the case of the machine learning approach, the same reinforcement learning (RL) technique that was previously developed by the Autonomic Computing Group at IBM is used [67, 68]. The RL algorithm has already been trained off-line before being used in these experiments. Therefore, we refer to this technique as Trained RL. Each run lasts for about 14 hours. For each run we drop the first 250 controller intervals as we expect them to include the transient period. Then, we consider only the next 10,000 controller intervals for each run.

5.3.5 Results

In this section, we present the results obtained for each set of experiments. For each set of experiments we compare the performance of the controller when servers allocation decisions are either random, static, based on the performance model in section 5.3.3 or the trained RL. For each run we collect results that correspond to 10,000 allocation decisions (after the transient period). The results obtained for these 10,000 control intervals are aggregated into 25 averaged results obtained for the 25 summarized intervals that we refer to as Aggregate Intervals. Therefore, the length of each aggregate interval is 400 times the duration of a controller interval (5 sec), that is a total length of 2000 seconds.

5.3.5.1 Results for The Fixed Number of Customers Case

The results reported in this section were obtained from experiments with a fixed number of customers equal to 45 and a varying think time. Figure 5.21 shows the averaged variations of the think time during these experiments.

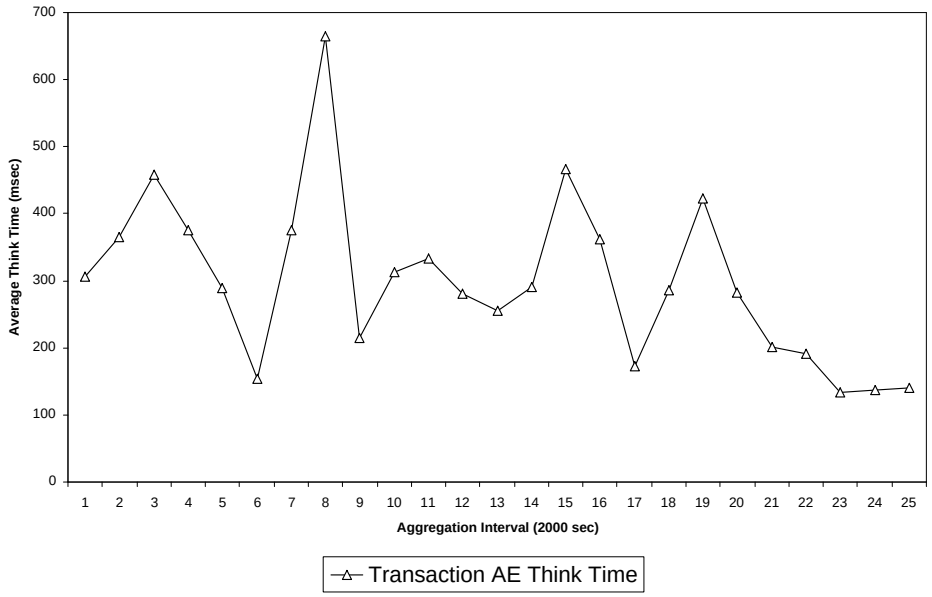


Figure 5.21: Variations for the average think time.

The variations in the average arrival rates that results from variations in the think time are shown in Fig. 5.22.

The server allocation decisions made through the experiments for the case of the online transaction AE are depicted in Fig. 5.23. Static allocation allocates 3 servers, random allocation allocates more than 3 servers during the run while the performance model and Trained RL both allocate less than 3 servers in general and get close to that in the last 3 aggregate intervals where the load peaks to 270-275 requests per

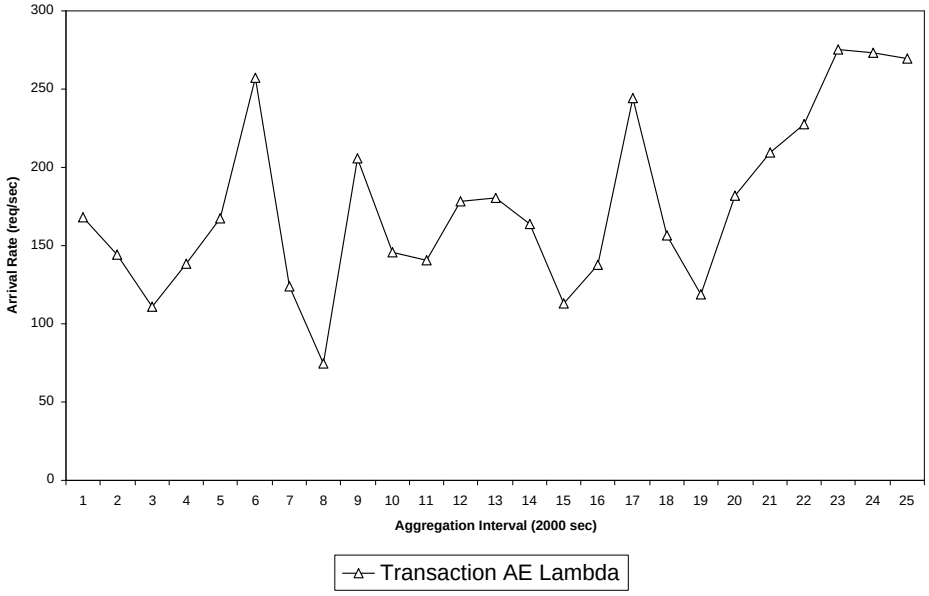


Figure 5.22: Variations for the average arrival rate.

second. The figure shows that trained RL consistently allocates less servers than the performance model while at the same time following a very similar allocation pattern.

In Fig. 5.24 we show the variations of the average response time for the online transaction AE. This figure shows that for most of the techniques, the measured response times are well below the target SLA except for some few points for the random allocation. The static allocation (3 servers) generally achieves the lowest response times except for the last 3 aggregate intervals. In these last 3 intervals, the average arrival rates peak to 270 and 275 requests per second. This observation suggests that for most of the run the transaction AE had sufficient computing capacity with 3 servers but would have needed some additional servers as the load was peaking for the last 3 aggregate intervals. This figure also shows that for most of the time the response time achieved by the performance model is lower than for the trained RL.

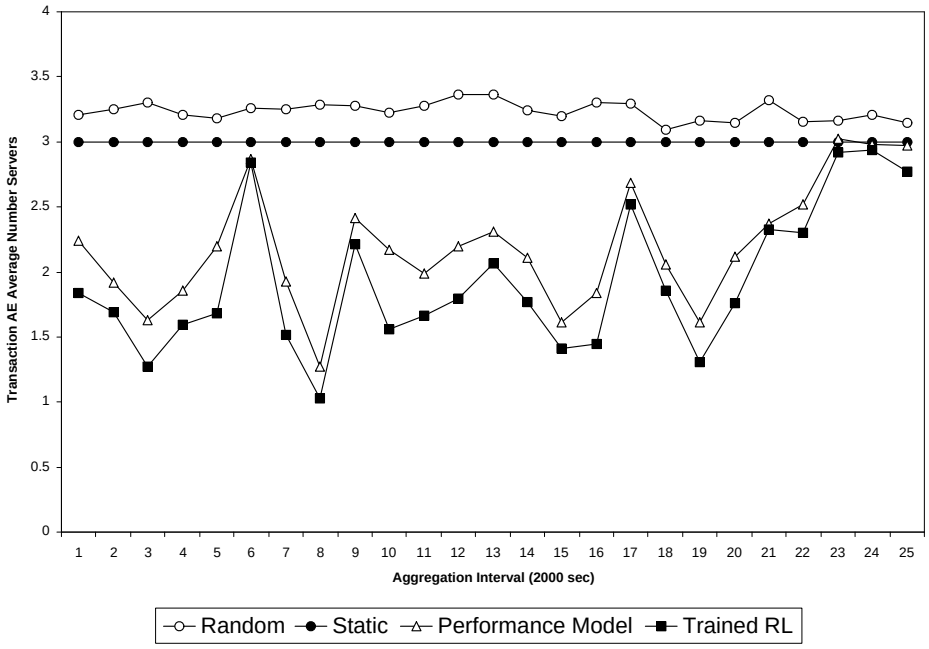


Figure 5.23: Variations for the average number of servers allocated to the transaction AE for a fixed number of customers.

These observations made regarding the response time are further reflected in the utility of the transaction AE curves in Fig. 5.25. For most of the run, the static and the performance model based allocations perform the highest except for the last 3 aggregate intervals when the achieved local utility for the static allocation drops. The random allocation performs the worst for several aggregate intervals with even a negative utility value of -2.35 at the last aggregate interval. Trained RL achieves a local utility curve that is lower than in the performance model case but again both curves have similar shapes.

Figure 5.26 shows the variations of the average local utility for the batch AE. The random allocation achieves the lowest local utility values while the static allocation yields a constant local utility value equal to 34 that corresponds to the two assigned

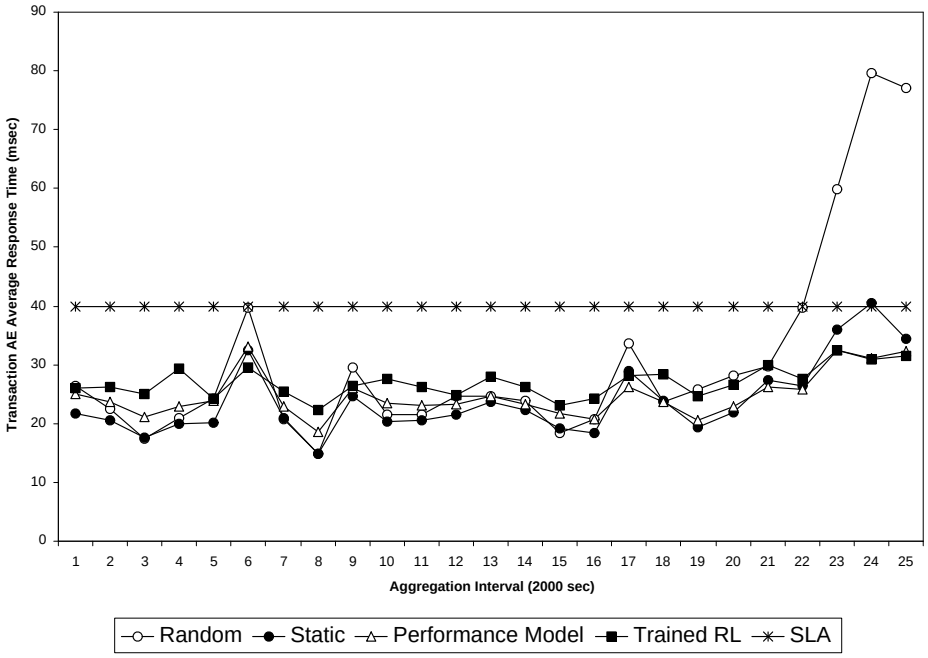


Figure 5.24: Variations for the average response time for the transaction AE for a fixed number of customers.

servers. Unlike the case of the transaction AE, trained RL achieves higher local utility values than in the case of the performance model. However, as shown in the figure, the two curves do have similar tendencies this time again.

The global utility values for the whole data center are shown in Fig. 5.27. As the figure shows, the random allocation performs the worst, followed by the static allocation. For the cases of the performance model and Trained RL, the two curves are very close to each other with some points where visibly performing better for one technique than the other. When conducting a paired t-test to compare the performance model to trained RL at a 95% confidence level, the obtained interval for the difference between the performance model and the trained RL utility values was [-2.865, 1.989]. This interval contains 0 and therefore no conclusion can be drawn.

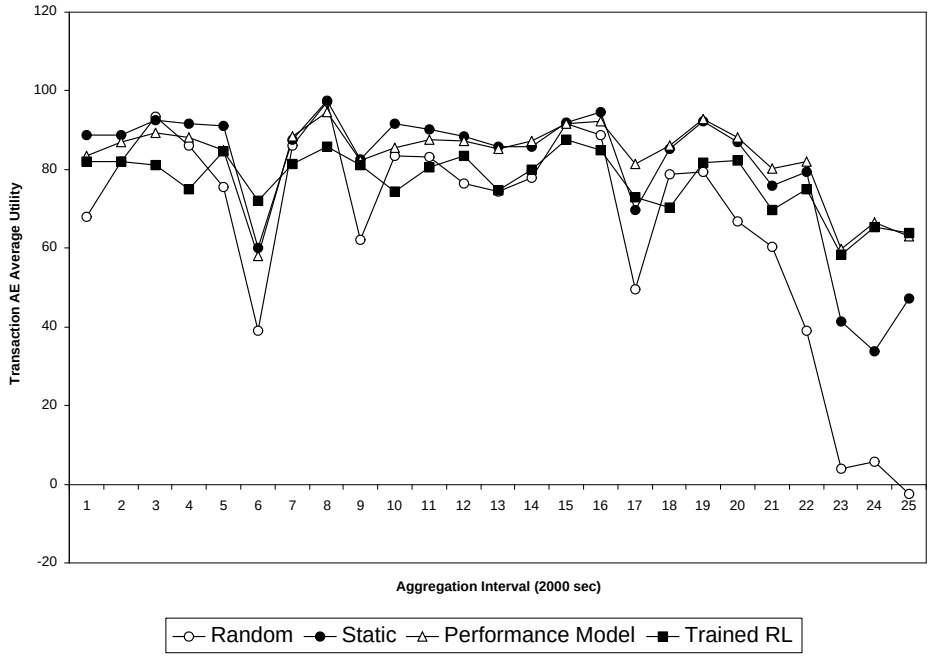


Figure 5.25: Variations for the average local utility for the transaction AE for a fixed number of customers.

5.3.5.2 Results for The Fixed Average Think Time Case

The results reported in this section were obtained from experiments with a fixed average think time equal to 167 msec and a varying number of customers between 10 and 60. Figure 5.28 shows the averaged variations of the number of customers during these experiments.

The workload applied in the experiments of this section is a bit higher than in the case of fixed number of customers. For this reason, the run for the static allocation was subject to the crash of one of the 3 servers allocated to the transaction AE in the last 30 minutes of the run. Therefore, the corresponding data points for the last aggregate interval are missing in the curves presented in this section.

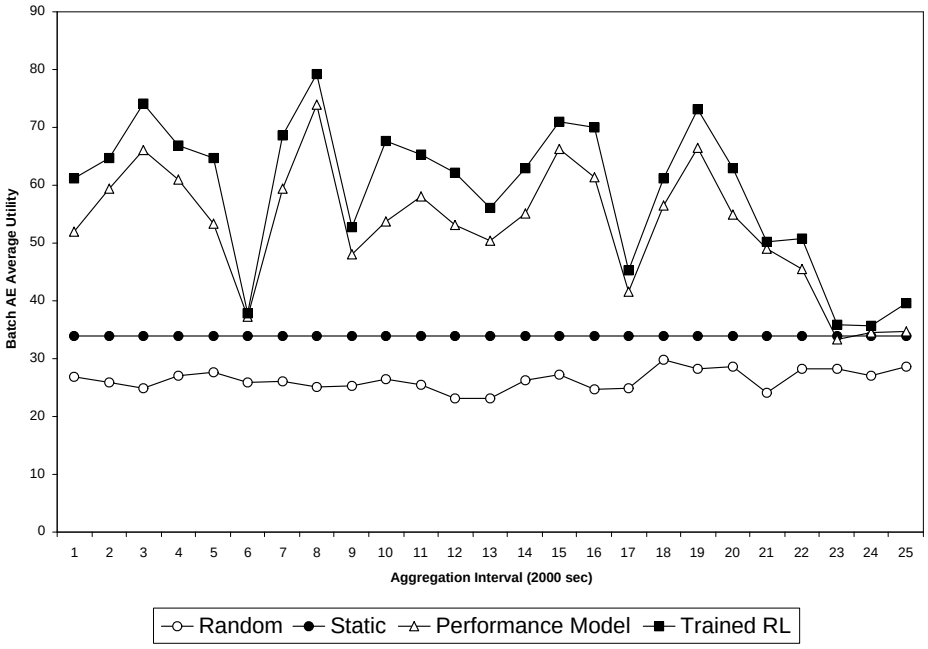


Figure 5.26: Variations for the average local utility for the batch AE for a fixed number of customers.

The server allocation decisions made through the experiments for the case of the online transaction AE are depicted in Fig. 5.29. Static allocation assigns 3 servers, random allocation assigns more than 3 servers during the run while the performance model and trained RL both allocate less than 3 servers for most of the run. However, the performance model allocates about 3.5 servers on average at aggregate interval 6 where the number of customers surges to 56. The performance model even allocates a fourth server at aggregate intervals 23-25 where the number of customers reach peak values of 60 and 58. Trained RL allocates the least number of servers (no more than 3) during the entire run. The figure also shows that trained RL and the performance model follow similar patterns for server allocation.

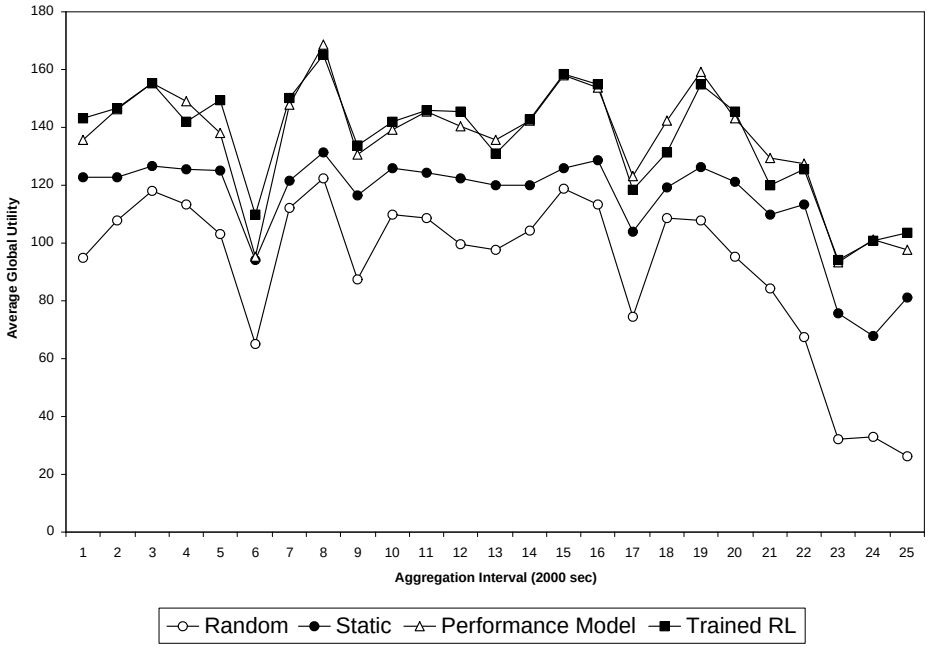


Figure 5.27: Variations for the average global utility for the data center for a fixed number of customers.

Figure 5.30 shows the variations of the average response time for the online transaction AE. The figure shows that the performance model achieves the lowest response time in general. For the case of the performance model, the response time is well below the SLA for the entire run; the highest value is 33 ms at aggregate interval 6. This is not the case for the other techniques. For the random allocation, the response time values exceed the SLA by far at many aggregate intervals with the largest value of 128 ms recorded at the peak load. For the static allocation, the lowest response time values are obtained only under low load. As the load reaches high levels (aggregate intervals 6, 17, 20-25), the response times go beyond the SLA. For trained RL, the response times are higher than in the performance model case but both curves do have similar variation tendencies. The response time values get around the SLA

Figure 5.28: Variations for the average number of customers.

when the load is at its peak for the trained RL case.

The observations made regarding the response time are again reflected in the utility of the transaction AE curves in Fig. 5.31. For most of the run, the performance model based allocations outperform other techniques; the lowest utility value of 61 was obtained at aggregate interval 6. While the trained RL curve keeps the same variation tendencies as in the performance model case, the obtained utility values, however, are much lower with the ever lowest value of 30 recorded for aggregate interval 23. Static allocation achieves high utility values only under light load. For high load conditions, the achieved utility values are very low and go below 0 starting at aggregate interval 23. The random allocation yields the lowest utility values.

Figure 5.32 shows the variations of the average local utility for the batch AE. The

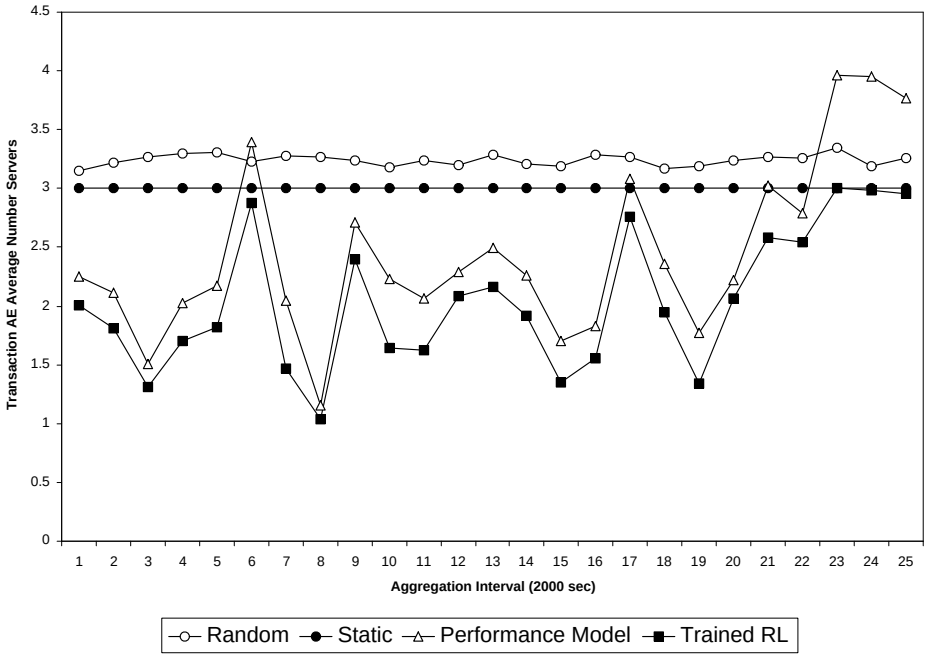


Figure 5.29: Variations for the average number of servers allocated to the transaction AE for a varying number of customers.

random allocation achieves the lowest local utility values (about 27 overall) while the static allocation yields a constant local utility value equal to 34 that corresponds to the two assigned servers. Unlike the case of the transaction AE, this time again trained RL achieves the highest local utility values. For the case of the performance model, the utility values are somehow lower than in the trained RL case except for the last three aggregate intervals where the utility is around 8 (only 1 server assigned to the batch AE by the performance model while the other 4 servers were assigned to the transaction AE because of its workload surge). However, as shown in the figure, the two curves do have similar tendencies, in general, this time as well.

The global utility values for the whole data center are shown in Fig. 5.33. As the

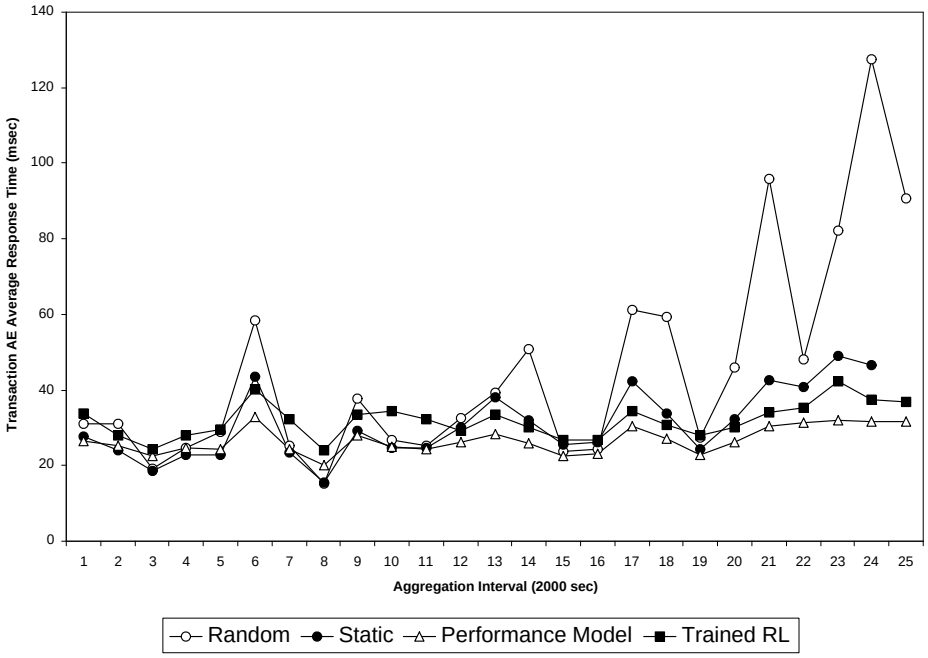


Figure 5.30: Variations for the average response time for the transaction AE for a varying number of customers.

figure shows, in this case again, the random allocation performs the worst and it is followed in that by the static allocation. For the cases of the performance model and trained RL, the two curves are close to each other with an apparent advantage for the performance model for the case of most points. To confirm this conclusion, we conducted a paired t-test to compare the performance model to trained RL at a 95% confidence level. The interval obtained for the difference between the performance model and the trained RL utility values was [4.081, 6.465]. This interval does not contain 0 and therefore we conclude that the performance model is superior to trained RL under this workload where the average number of customers is varying and the average think time is kept constant.

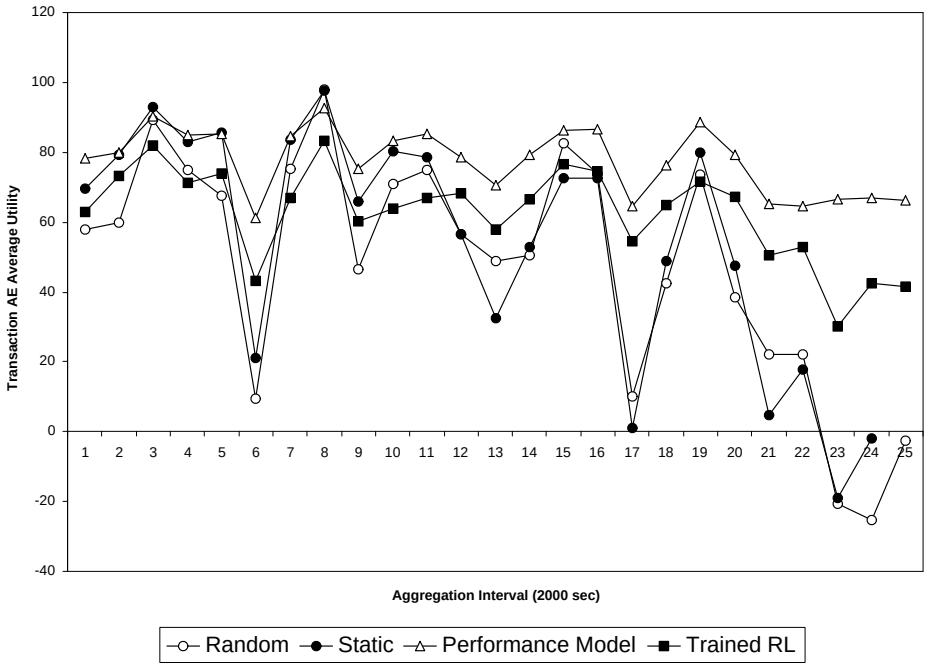


Figure 5.31: Variations for the average local utility for the transaction AE for a varying number of customers.

5.4 Comparison of Machine Learning and Analytic Performance Model Based Controllers

In this section we provide a comparison between controllers that use a machine learning technique or an analytic performance model to drive their resource allocation decisions.

The first fundamental difference between these two radically different approaches is that machine learning control falls into the category of black box control along with a control theoretic approach, for example. On the other hand, control that is based on analytic performance model falls into the category of white box control. The main advantage of black box control is that no knowledge of the internal components of

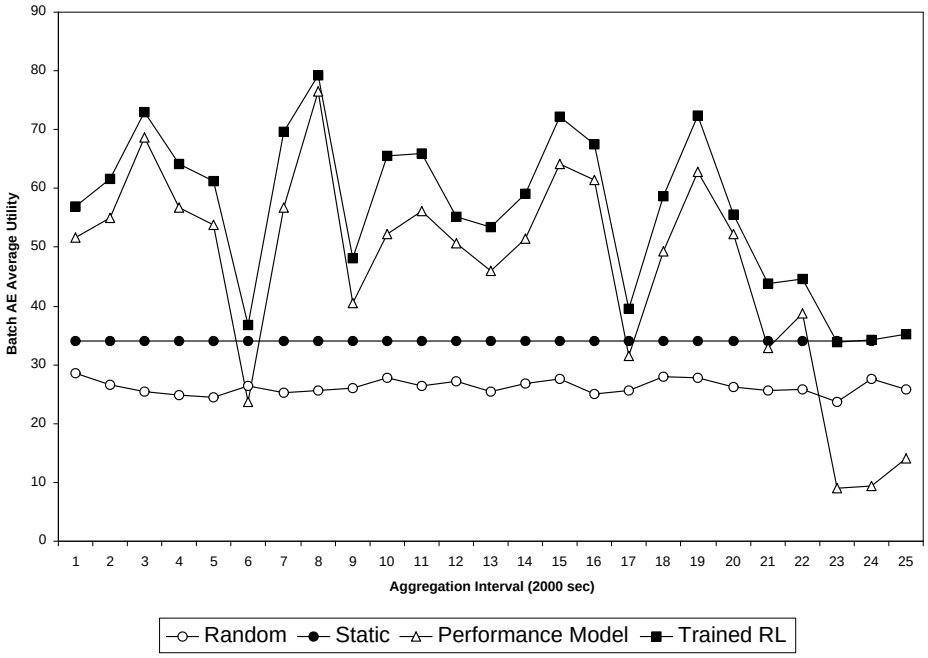


Figure 5.32: Variations for the average local utility for the batch AE for a varying number of customers.

the system is required. The controller simply tries to infer relationships between the inputs and outputs of the system. This gives an edge to a machine learning approach for systems for which parameter inter-dependencies are not clear, not much knowledge about system under test is available, or the performance model is very difficult to solve or its input paramaters are not easily obtainable.

On the other hand, a machine learning approach requires some significant training time before becoming ready for deployment. Besides that, any changes in the system in terms of new or upgrade of physical resources, new application environments, changes in the SLAs, or even changes in the workload patterns would require the machine learning algorithm to be re-trained. This lack of flexibility could significantly hinder the wide adoption of this approach in the self-management of today's complex

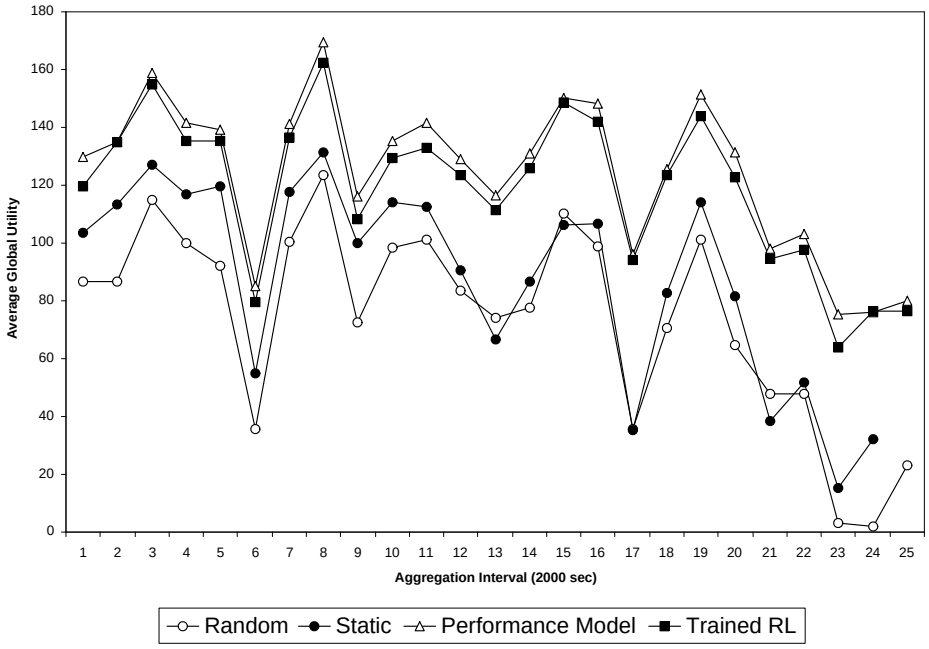


Figure 5.33: Variations for the average global utility for the data center for a varying number of customers.

computing systems due to their very dynamic nature. Also, it is not clear yet how efficient a machine learning approach could be for the cases of application environments with multiple classes. The work conducted in [67, 68] deals only with a single class of customer requests per application environment for the moment.

5.5 Availability and Performance in Autonomic Data Centers

This section shows how autonomic computing techniques can be used to dynamically allocate servers to application environments in a way that maximizes a global utility function for the data center even when servers fail.

5.5.1 The Experimental Setting

In the experiments of this section we consider a simulated internet data center with two online AEs. AE 1 has three classes of transactions and AE 2 has two transaction classes. The total number of servers in the data center is 30 and each server is assumed to have one CPU and one disk.

Two scenarios are considered:

- *Server failure with fixed workload intensity*. In this scenario, one of the servers fails at time unit 5 and recovers at time unit 15. The average workload intensity is maintained constant for all classes as $\vec{\lambda}_1$ = (20 tps, 40 tps, 14 tps) and $\vec{\lambda}_2$ = (120 tps, 240 tps). The purpose of this experiment is to show how the controller is capable of moving servers between the AEs in response to failure and recovery events.

- *Server failure with variable workload intensity*. In this scenario the workload intensity varies as well and one server fails as in the previous case.

Table 5.3 shows the service demands (in sec) for the CPU and disk, the SLA per class (in sec), as well as the weight of each class for the computation of the utility function of the AE. The global utility function used in all experiments is $U_g = 0.5 \times (U_1 + U_2)$.

5.5.2 Results

The following subsections show the numerical results obtained for the two scenarios described earlier. Each point is an average over 10 runs with 95% confidence intervals shown. In many cases, the confidence intervals are so small that the bars are not

Table 5.3: Input Parameters for the Experiments

Application Environment 1			
s	1	2	3
$D_{1,s}^{\mathrm{CPU}}$	0.030	0.015	0.045
$D_{1,s}^{\mathrm{IO}}$	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			
s	1	2	
$D_{2,s}^{\mathrm{CPU}}$	0.030	0.015	
$D_{2,s}^{\mathrm{IO}}$	0.024	0.010	
$\beta_{2,s}$	0.100	0.050	
$a_{2,s}$	0.450	0.550	

visible. The x-axis on all graphs shown in the following subsections indicate the progression of time during the experiment measured in control intervals, which are 2 minutes each.

5.5.2.1 Fixed Workload Intensity with Failures

In the curves shown in this section, the average arrival rates are fixed for each class. However, since a server allocated to AE 2 fails at time equal to 5 units, the global utility function of the non-controlled case shows a clear drop at that time as shown in Fig. 5.34. When the server recovers at time 15, U_g increases to its initial level. Note that the controller is able to maintain U_g pretty much constant despite the server failure. It should be emphasized that the small variations of U_g are due to the stochastic nature of the arrival process. What is fixed in these experiments is the average arrival rate.

The variation of number of servers is illustrated in Fig. 5.35. Without the controller, n_1 starts at 14 and n_2 at 16. At time 5, n_2 goes to 15 and remains at that

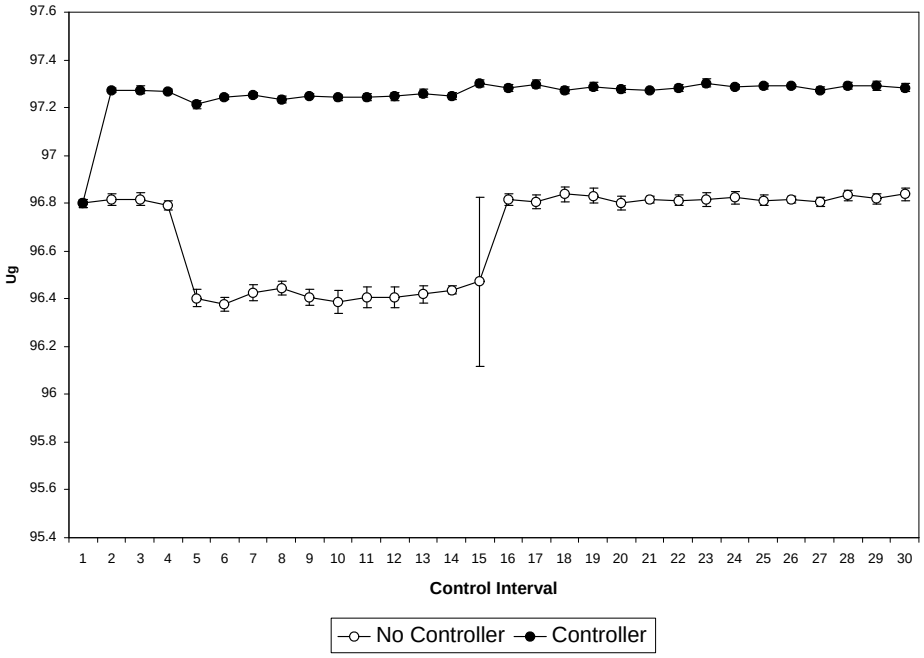


Figure 5.34: U_g for Fixed Arrival Rate and Failures.

values until time 14. At time 15, n_2 returns to 16. When the controller is used, n_1 starts at 14 but changes immediately to 10 at time 2. The value of n_2 starts at 16 but moves immediately to 20 at time 2. When a server of AE 2 fails at time 5, n_2 goes to 19, but the controller, takes one server away from AE 1 at time 6 so that n_2 can return to 20. When the failed server recovers at time 15, AE 1 regains the server it had lost due to the failure of a server allocated to AE 2.

Figure 5.36 shows the response time for class 1 of AE 1 for the controller and no-controller cases. The figure shows that the controller makes the SLA for this class to be slightly violated by 6.7% of its target value of 0.06 sec. This small loss in response time is counterbalanced by a significant gain of 66.7% in the response time of class 1 of AE 2. This gain is reflected as well in the total utility value as shown in Fig. 5.34.

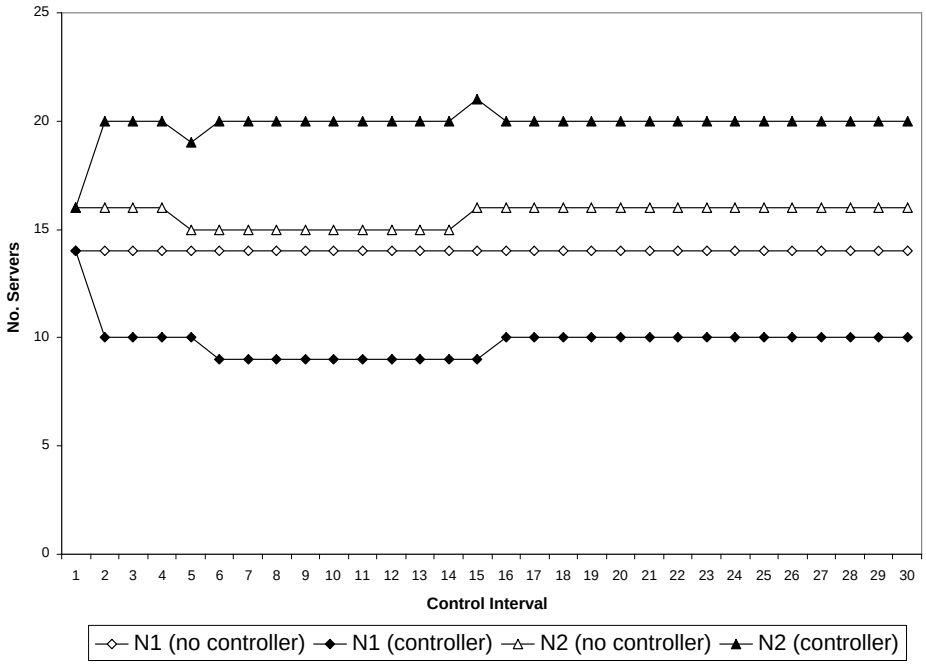


Figure 5.35: Variation of Number of Servers for Fixed Arrival Rate and Failures.

Figure 5.37 shows the variation of the response time for class 1 of AE 2. It can be seen that when the controller is not used, the response time increases at the moment the server fails and only decreases when the server recovers. The controller manages to keep the response time virtually constant due to the dynamic server reallocation.

5.5.2.2 Variable Workload Intensity with Failures

The purpose of this experiment is to illustrate how the controller is capable of dynamically moving servers between the two AEs to cope with both the cases of a variation in the workload intensity and servers failure. Figure 5.38 shows the variation of the workload intensity for all five classes during the experiment. On purpose, the peaks in workload intensity for AE 1 coincide with the valleys for AE 2 and vice-versa.

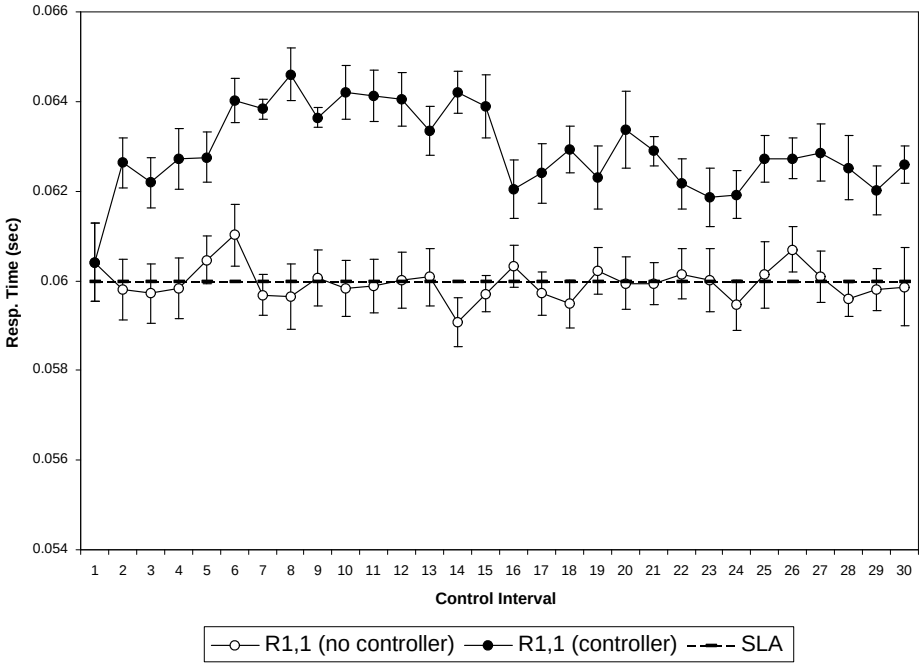


Figure 5.36: Variation of the Response Time for Class 1 of AE 1 for Fixed Arrival Rate and Failures.

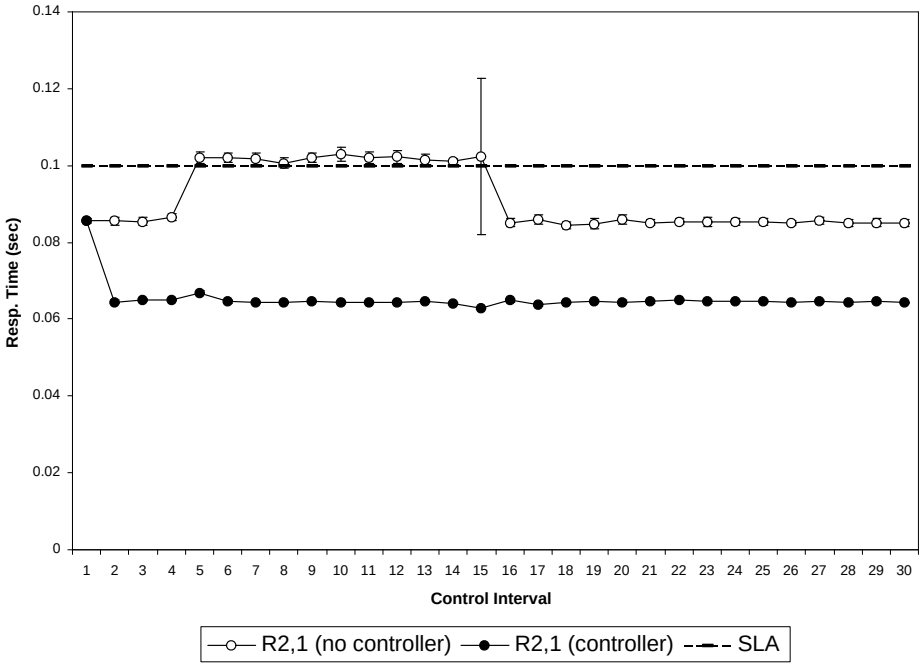


Figure 5.37: Variation of the Response Time for Class 1 of AE 2 for Fixed Arrival Rate and Failures.

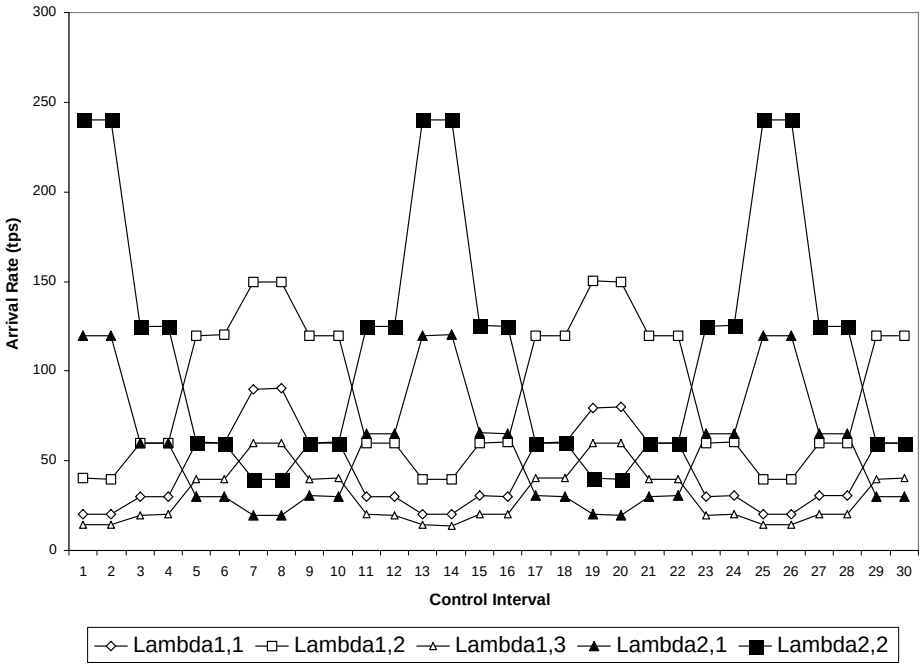


Figure 5.38: Variation of the arrival rate for all classes of AEs 1 and 2.

Again in this experiment, a server allocated to AE 2 fails at time unit 5 and recovers at time unit 15.

Figure 5.39 depicts the total utility function U_g. As it can be seen, the controller is able to cope with the simultaneous variability in the workload intensity and server failure and maintain U_g pretty much constant.

The curves for number of servers, Fig. 5.40, illustrate how servers are switched among AEs to give each of them the number of servers needed to bring the utility function to its highest possible level for each circumstance.

Figure 5.41 shows the variation of the response time for class 1 of AE 1. The figure shows that if the controller is not used, the response time for this class reaches peaks that are much higher than the SLA for that class (i.e., 0.06 sec) at time units

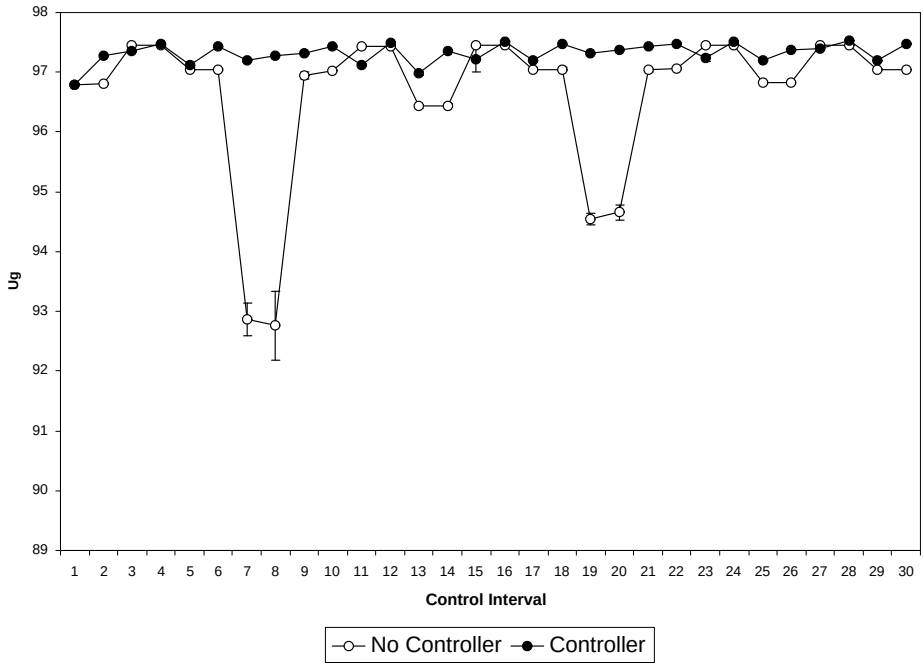


Figure 5.39: U_g for Variable Arrival Rate and Failures.

7 and 19. These instants correspond to the peak arrival rates for this class. However, when the controller is used, the response time stays close to the SLA even when the the workload intensity surges for these classes.

The response time for class 1 of AE 2 is shown in Fig. 5.42. In this case, the difference between the controller and no-controller situations is not as marked as in Fig. 5.41 because the SLA is not as strict for this class as is the case with the classes of AE 1. As it can be seen in Fig. 5.42, the response time is generally below the SLA of 0.10 sec, except for time units 12 and 13 where the workload intensity surges for class 1 of AE 2 . There are even some cases in which the response time with the controller is worse than that without the controller. This happens because the controller is giving more servers to AE 1 because of its more stringent response time

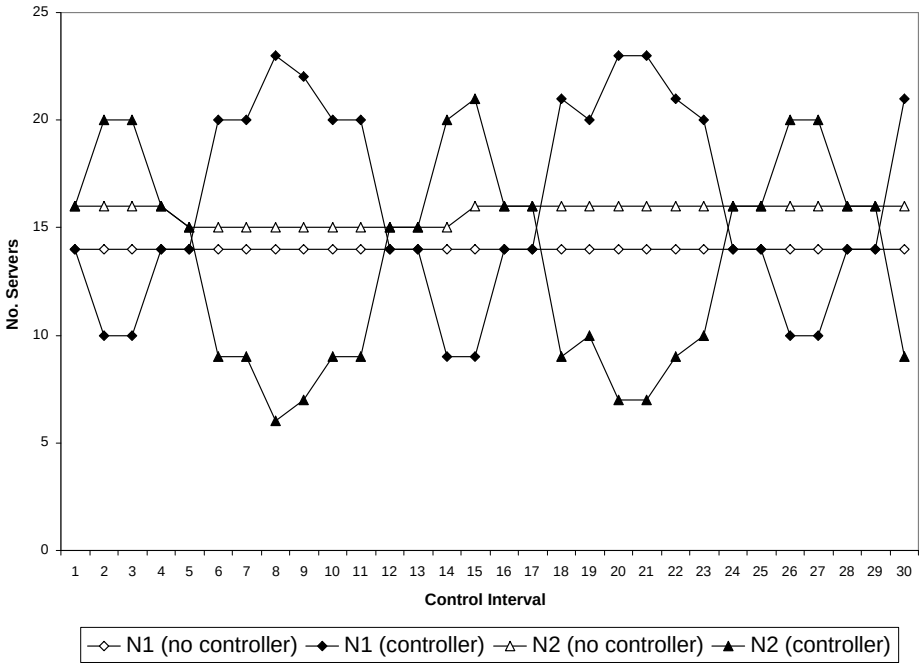


Figure 5.40: Variation of Number of Servers for Variable Arrival Rate and Failures.

requirements.

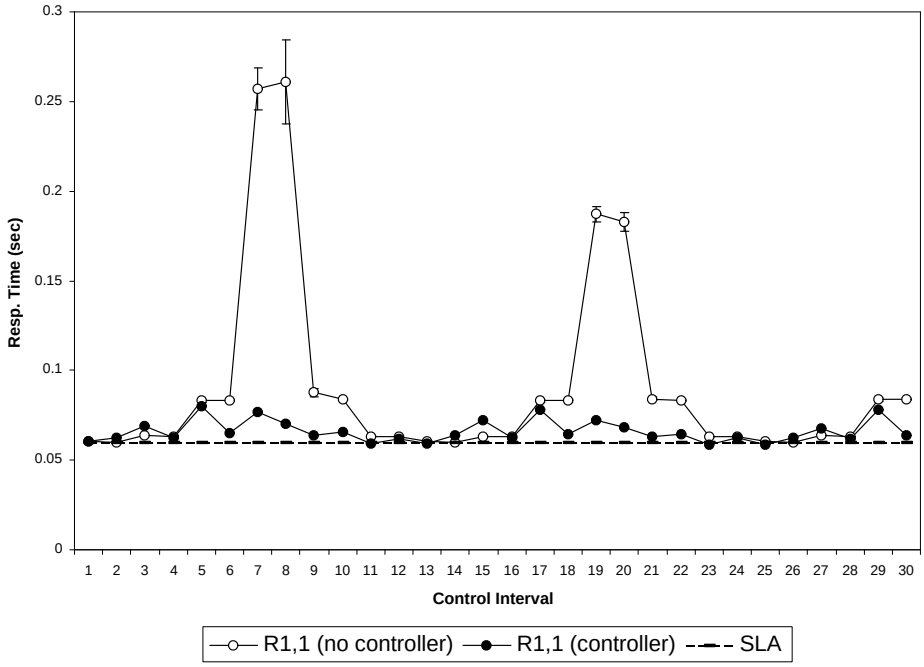


Figure 5.41: Variation of the Response Time for Class 1 of AE 1 for Variable Arrival Rate and Failures.

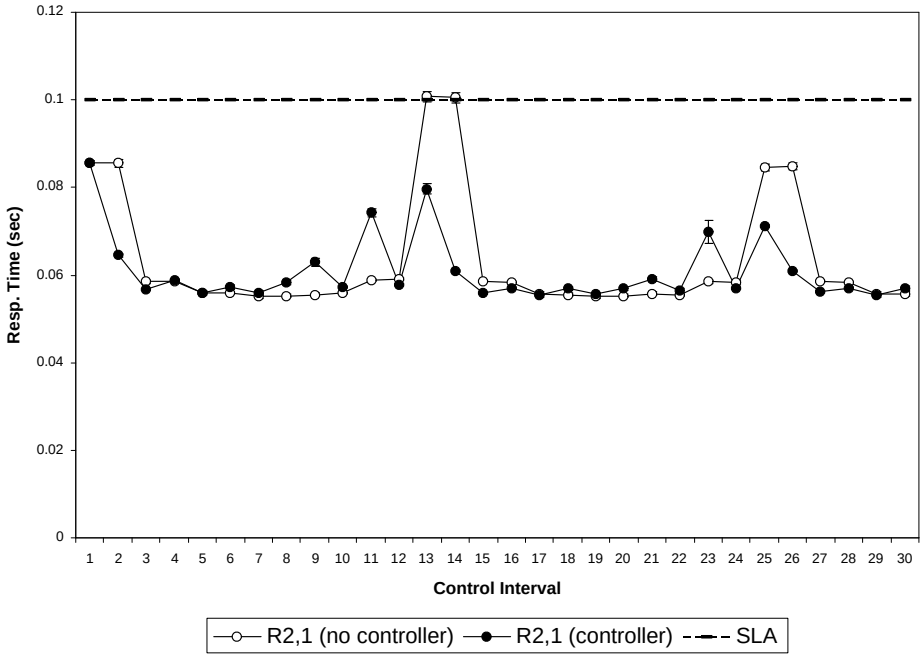


Figure 5.42: Variation of the Response Time for Class 1 of AE 2 for Variable Arrival Rate and Failures.

Chapter 6: Autonomic Computing For Virtualized Environments

This chapter deals with autonomic computing for virtualized environments. It studies how online analytic performance models could be used to enable virtualized systems to manage their configurations dynamically so as to increase their overall performance. Virtualization offers several advantages in terms of an increased security, a reduced total cost, and an easy support for legacy applications [49]. For these reasons, virtualized environments are back gaining ground in organizations' computing facilities. In fact, several recent commercial products provide virtualization capabilities. These include Solaris 10, IBM LPAR, XEN, and VMWare. However, because of the increased unpredictability in nowadays workloads, virtualized environments need to be autonomic in nature in order to deliver adequate performance.

This chapter deals with the problem of dynamic CPU allocation in virtual servers. The chapter covers two techniques for CPU virtualization. The first technique, priority based allocation, assigns the CPU to applications according to their specified priorities. The second technique, referred to as CPU share based allocation, assigns only some of the CPU shares to an application at any time. The number of CPU shares can vary dynamically in order to maximize the global utility value of the virtual server. In this chapter, the mathematical formulation of the problem uses the same framework that appears in the previous two chapters (Autonomic Multithreaded Servers and An Autonomic Data Center) and adapts it to the case of an autonomic

virtual server.

This chapter is organized as follows. The first section formally defines the problem of dynamic allocation of the CPU. The second section presents the controller approach. The performance models are presented in section 3. The experimental setting and results are reported in sections 4 and 5, respectively.

Work presented in this chapter appears in the following publication: [56].

6.1 The Dynamic CPU Allocation Problem

We consider a virtualized environment (virtual server in this case) with M virtual machines (VM). Each VM i may run S_i different workload classes. We consider in this chapter the problem of dynamically allocating the CPU to the various virtual machines in response to changes in the workload intensity. The workload intensity level for workload class s at VM i is specified by the average arrival rate $\lambda_{i,s}$ of transactions of class s at VM i. We define a workload vector $\vec{w}_i$ for VM i as $\vec{w}_i = (\lambda_{i,1}, \cdots, \lambda_{i,S_i})$. We consider in this work that the relevant performance metric for the workload classes are the average response times $R_{i,s}$ for each class s of VM i. Including throughput as a performance metric of interested is straightforward. We define a response time vector, $\vec{R}_i$, for VM i as $\vec{R}_i = (R_{i,1}, \cdots, R_{i,S_i})$. Again these performance metrics can be obtained by solving an analytic performance model, $\mathcal{M}_i$, for VM i. The value of these metrics is a function of the workload $\vec{w}_i$ and of the CPU allocation, a_i, to VM i. Thus, $\vec{R}_i = \mathcal{M}_i(\vec{w}_i, a_i)$. We investigate two approaches for assigning the CPU allocation a_i to each VM.

- *Priority allocation*: Each VM i has a dispatching priority, p_i ($p_i = 1, ..., P$), at the CPU. We assume preemptive resume priorities with priority 1 being the highest and priority P being the lowest. The priority changes dynamically according to the autonomic controller.

- *CPU share allocation*: Each VM i is allocated a share f_i of the CPU ($\sum_{i=1}^{M} f_i = 1$). VMWare allows CPU shares to be allocated to different virtual machines in order to influence their processing rate (www.vmware.com). For example, if VM1 has 1,000 CPU shares and VM2 and VM3 have 500 CPU shares each, then VM1 will execute at a rate twice as high as VM2 and VM3, which will execute at the same rate. In the case of VMWare, the allocation of CPU shares can be changed by an administrator. In this chapter we show how this can be done by the autonomic controller.

Each VM i has a utility function U_i that is a function of the set of performance metrics for the classes of that VM. So, $U_i = f(\vec{R}_i)$. The global utility function, U_g, of the entire virtualized environment is a function of the utility functions of each VM. Thus,

$$U_g = h(U_1, \cdots, U_M). \tag{6.1}$$

The utility function that we use in this chapter for class s at VM i, $U_{i,s}$, is defined as the relative deviation of the response time $R_{i,s}$ of that class with respect to its service level agreement (SLA), $\beta_{i,s}$. This utility function is obtained from the QoS function defined in chapter 3 by considering only the response time component. Again, any

other function could have been considered instead. Hence,

$$U_{i,s} = \frac{\beta_{i,s} - R_{i,s}}{\max\{\beta_{i,s}, R_{i,s}\}}. \tag{6.2}$$

If the average response time $R_{i,s}$ meets the SLA, $U_{i,s}$ is equal to zero. If $R_{i,s} > \beta_{i,s}$, then $-1 < U_{i,s} < 0$. If $R_{i,s} < \beta_{i,s}$, then $0 < U_{i,s} < 1$. Thus, the utility function $U_{i,s}$ is a dimensionless number in the $(-1, 1)$ interval. Other utility functions can be defined. See [11, 71] for examples. The utility function for VM i, U_i, is a weighted sum of the class utility functions. So,

$$U_i = \sum_{s=1}^{S_i} \alpha_{i,s} \times U_{i,s} \tag{6.3}$$

where $0 < \alpha_{i,s} < 1$ and $\sum_{s=1}^{S_i} \alpha_{i,s} = 1$. U_i is also a number in the $(-1, 1)$ interval. The resource allocation problem in question is how to dynamically allocate the CPU among VMs with the goal of maximizing the global utility function U_g. The dynamic CPU allocation is achieved through an autonomic controller of the *white box* type; i.e., a controller that knows the internal details of the system being controlled and can use this knowledge to build a model that relates the output with the input for a given value of the controllable parameter. In this chapter, we develop specific analytic models to be used by the controller in this virtualized environment.

6.2 The Controller Approach

The goal of the autonomic controller is to find an optimal CPU allocation vector $\vec{a} = (a_1, \cdots, a_M)$ to the set of M virtual machines that optimizes the global utility function U_g. Let $\mathcal{S}$ be the space of all possible allocations. In the case in which the

CPU allocation is priority-based, the cardinality of $\mathcal{S}$ is P^M where P is the number of possible priority levels for the virtual machines. If $P = M = 10$, there are 10 billion possible CPU allocations. In the case of CPU-share allocations, we discretize $\mathcal{S}$ by assuming that the total number of shares is T and that a VM is given a number of shares $n_i,\ n_i = 1, 2, \cdots, T$ such that $\sum_{i=1}^{M} n_i = T$. Thus $f_i = n_i/T$. It is not difficult to see, using combinatorial arguments, that the cardinality of $\mathcal{S}$ in the CPU share case is given by

$$\mid \mathcal{S} \mid = \left(\begin{array}{c} T + M - 1 \\ M - 1 \end{array} \right) - \sum_{j=T+1}^{T+M-2} j \tag{6.4}$$

which can be written as

$$\left(\begin{array}{c} T + M - 1 \\ M - 1 \end{array} \right) - \frac{(M-2)}{2}\ (2\ T + M - 1). \tag{6.5}$$

For T = 100 and M = 10, the number of possible allocations is a very large number in the order of 10^{13}. As it can be seen, the space of possible allocations $\mathcal{S}$ can be extremely large depending on the values of P, M, and T. The autonomic controller uses heuristic combinatorial search techniques [63], beam-search in our case, to explore a subset of $\mathcal{S}$ in which a solution close to the optimal may be found. Clearly, if the size of $\mathcal{S}$ is small enough, an exhaustive search should be conducted by the controller. The utility function associated with each point in $\mathcal{S}$ is computed using the performance models discussed in Section 6.3. Here again, at the end of every controller interval (CI), the controller runs the controller algorithm, which searches through a subset of $\mathcal{S}$, as explained above, and determines the best allocation $\vec{a}$, which is communicated to the CPU scheduler module of the VMM.

More specifically, the controller algorithm searches the space of CPU allocations $\vec{a} =$

$(a_1, \cdots, a_M)$ using a beam-search algorithm as follows. Starting from the current allocation $\vec{a}_0$, the global utility function, U_g, is computed for all the "neighbors" of that allocation. The k allocations 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 to more formally define the concept of neighbor of an allocation. Let $\vec{v} = (v_1, v_2, \cdots, v_M)$ be a *neighbor* of an allocation vector $\vec{a} = (a_1, a_2, \cdots, a_M)$. In the priority case, $\vec{v}$ and $\vec{a}$ represent priorities of the M VMs in the range $1, \cdots, P$. The allocation vector $\vec{v}$ is a neighbor of $\vec{a}$ if the priority of one of the VMs in $\vec{v}$ differs from the priority of the same machine in $\vec{a}$ by $\pm$ 1 in the sequence $1, \cdots, P$. Adding one to priority P should result in priority 1 and subtracting one from priority 1 should result in priority P. In the case of CPU-share allocations, the allocation vector $\vec{v}$ is a neighbor of $\vec{a}$ if $v_i = a_i + 1/T$ and $v_j = a_j - 1/T$ for some $i \neq j$ and $v_k = a_k$ for all $k \neq i$ and $k \neq j$. The controller algorithm is specified more precisely in Figure 6.1. The following notation is in order:

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

- $\vec{a}_0$: current allocation vector.

6.3 The Performance Models

In this section we present the performance models used by the controller to predict the performance of various allocation configurations. We distinguish between the cases of whether allocations are based on the priority of the virtual machines or on their respective CPU shares.

The queuing network models used here fall into the category of open multiclass queuing network models [42]. Each queue in the queuing network (QN) represents a resource (e.g., CPU, disk). There may be multiple classes of transactions depending on their use of the resources and their workload intensities. The total time spent by a transaction class s using a resource k is its service demand, $D_{k,s}$. It is to be recalled that the service demand does not include any queuing time spent at the resource. We consider in this chapter that all workloads are open workloads whose intensities are specified by their arrival rates. The response time, $R_{i,s}$, for class s of VM i is given by [42]:

$$R_{i,s} = \sum_{k=1}^{K} \frac{D_{k,s}}{1 - \sum_{s=1}^{S_i} \lambda_{i,s} \times D_{k,s}} \tag{6.6}$$

where K is the number of resources used by transaction s of VM i. Equation (6.6) does not cover the case of priority-based scheduling nor the case of CPU shares. In the next subsections we show how we adapt the results of Eq. (6.6) to these situations.

6.3.1 Performance Model for Priority Based Allocation

To model CPU preemptive resume dispatching priorities, we use the shadow CPU approximation in [42]. All the workloads of the same VM have the same priority in the model, since the VMM assigns dispatching priorities to the VM as a whole and not to its individual workloads. In this approach, the open QN model is solved incrementally in P steps, one per priority class, from the highest priority class 1 to the lowest priority class P. At step p, only classes of priority $q \leq p$ are included and the open QN model is solved. This partial model at step p has p shadow CPUs instead of one. Each shadow CPU is dedicated to all workloads that have the same priority. Let $D^p_{\text{CPU},s}$ be the CPU service demand for workload class s at the shadow CPU associated to priority p. For a workload s of priority $q \neq p$, $D^p_{\text{CPU},s} = 0$ since workload s only uses the CPU associated to priority q. For a workload s of priority p, one has to inflate the service demand at the dedicated CPU for that class to account for the fact that class s only sees the amount of CPU not used by all workloads of higher priority, i.e., priorities $q < p$. Thus,

$$D^p_{\text{CPU},s} = \frac{D_{\text{CPU},s}}{1 - \sum_{q=1}^{p-1} \sum_{r \in \Omega(q)} \lambda_r \times D_{\text{CPU},r}} \tag{6.7}$$

where $D^p_{\text{CPU},s}$ is the inflated CPU service demand for class s at the shadow CPU associated to priority p, $D_{\text{CPU},s}$ is the original CPU service demand of workload s, and $\Omega(q)$ is the set of workloads with priority q. The summation in the denominator of Eq. (6.7) is the sum of the CPU utilizations due to all workloads of priority $q < p$. Even though Eq. (6.7) does not explicitly indicate a VM, the workload designation s is assumed to be associated with a specific VM. The disk service demands do not

need to be changed because priorities only apply to the CPU. The final response time is obtained with the solution of the complete QN model at step P.

6.3.2 Performance Model for CPU-Share Based Allocation

The CPU share model is built by having M shadow CPUs, one per VM. The CPU service demands have to be adjusted to account for the share allocation of each VM. This is done by setting

$$D^e_{i;\mathrm{CPU}_s} = D_{i;\mathrm{CPU}_s}/f_i \tag{6.8}$$

where $D^e_{i;\mathrm{CPU}_s}$ is the elongated CPU demand of class s at the shadow CPU for VM i and $D_{i;\mathrm{CPU}_s}$ is the original CPU demand of class s of VM i. Equation (6.8) can be understood by interpreting f_i as the rate at which workloads of VM i execute. For example, if $f_i = 0.25$, then any workload of that VM will execute at a rate equal to 0.25 seconds of work per second of real time, assuming that a workload that receives a 100% allocation of the CPU executes at a rate of 1 second of work per second of real time. Thus, the average amount of time needed to execute one second of work is $1/0.25 = 4$ seconds in this example.

6.4 The Experimental Setting

We used simulation to evaluate the controller for the two different cases discussed before: priority and CPU shares. A CSIM (www.mesquite.com) model was built to simulate the virtualized environment in each case. The controller was implemented in a separate machine from the one used to run the simulation and used a controller interval equal to two minutes. For the simulation of the priority based case, we used

a preemptive resume scheduling discipline at the CPU with a time slice equal to 0.001 sec. For the CPU share case, we used a round-robin scheduling discipline at the CPU with a time slice of 0.001 sec. In both cases, the disk scheduling discipline is FCFS.

The experiments reported in what follows assume two virtual machines 1 and 2 with one workload class each. The methodology described above can be easily applied to a large number of virtual machines and workloads given that the utility function is computed using a very fast analytic model and the beam-search algorithm can be tuned by setting its breadth and depth parameters in order to limit the subset of $\mathcal{S}$ examined at each control interval. The service demands assumed in the experiments reported here are given in Table 6.1. As it can be seen, the I/O service demands were set at a very low value since the controller is acting on CPU allocation only. For this reason, we did not want the I/O activity to mask the results obtained in the experiments. We assume in the simulation experiments that CPU shares can only

Table 6.1: Service demands (in sec)

	Class	
	1	2
CPU	0.020	0.010
IO	0.0005	0.0005

vary in increments of 5%. The global utility function for the virtualized environment is computed as $U_g = 0.4 \times U_1 + 0.6 \times U_2$. All experiments reported here include a 95% confidence interval illustrated by the bars in each point in the graphs.

6.5 Results

In this section we present the simulation results obtained for each of the CPU allocation modes.

6.5.1 Results for the Priority Allocation

The variation of the workload intensity, in transactions per second (tps), for each of the workloads in each VM is shown in Fig. 6.2. The workload intensities show peaks and valleys, which are out of phase for the two VMs to force an exchange of CPU allocation between the two VMs. Workload 1 has its peaks at CIs 7 and 8 and then again at 19 and 20. Workload 2 has three peaks: at 1–3, 12, and 24.

Figures 6.3 and 6.4 show the variation of the utility functions U_1 and U_2 during the 60 minutes experiment, using the variation of the workload intensity shown in Fig. 6.2. Each curve shows the variation of the utility function when the autonomic controller is used and when it is not used.

The graph of Fig 6.3 shows that (1) there is very little difference between the controller and non-controller results, (2) the non-controller results seem to be slightly better in some cases especially when the workload of VM 2 has its peaks; this is due to the fact that the controller is providing more CPU resources to workload 2, which has a higher weight in the global utility function U_g.

Figure 6.4 shows a marked difference between the controller and non-controller cases. The utility function for U_2 remains in the positive territory from CI = 2 onwards.

Figure 6.5 shows the the variation of the global utility function U_g during the 60 minutes experiment. The figure shows that the controller is able to maintain a

significantly higher global utility for the virtualized environment even at periods in which the two workloads go through their peaks in workload intensity.

Figure 6.6 shows the variation of the priorities of VMs 1 and 2 during the 60-minute experiment. Both VMs start with the same priority (i.e., priority 2). The priority of VM1 remains unchanged throughout the experiment. However, VM2 receives an immediate priority boost to priority 1 (the highest) because this workload is at its workload peak. However, the controller reduces VM 2's priority to 2 (i..e., now both VMs have the same priority) when VM 1 goes through workload peaks. This demonstrates that the autonomic controller is able to react to changes in workload intensity in order to improve the global utility function.

Figures 6.7 and 6.8 show the variation of the response time for VMs 1 and 2, respectively, for the controller and non-controller cases, as well as the SLA (dashed line) for each case. For VM1, the response time for both cases (controller and non-controller) stays below the SLA of 0.8 sec except when the workload for VM 1 goes through its peak period. In these intervals, the controller and non-controller results have very little statistical difference as can be seen by the confidence intervals.

For the case of VM 2, Fig. 6.8 shows that as soon as the controller starts to operate (at CI = 2), the response time goes below its SLA of 0.035 sec because of the immediate boost in the priority level. The figure also shows that without the controller, this workload always exceed its SLA target.

6.5.2 Results for the CPU-Share Allocation

The variation of the workload intensity, in tps, for each of the workloads in each VM is shown in Fig. 6.9. As before, the workload intensities show peaks and valleys,

which are out of phase for the two VMs to force an exchange of CPU allocation between them. Workload 1 has its peaks at CIs 7–8, 19–20, and then again at 27–30. Workload 2 has three peaks: at 1–3, 12, and 24. Please note that although the shape of the curves in Fig. 6.9 is similar to those of Fig. 6.2, the workload intensity values in this former case is lower than in the latter.

Figures 6.10 and 6.11 show the variation of the utility functions U_1 and U_2 during the 60-minute experiment, using the variation of the workload intensity shown in Fig. 6.9. Each curve shows the variation of the utility function when the autonomic controller is used and when it is disabled. Figure 6.10 shows that U_1 is higher for VM 1 when the controller is not used. This is due to the fact the controller forces a higher level for U_2, as seen in Fig. 6.11, because VM 2 has a higher weight (0.6 versus 0.4) in the global utility function U_g.

Figure 6.12 shows the the variation of the global utility function U_g during the 60-minute experiment. The figure shows that in most cases the global utility function for the controller is significantly higher when the controller is enabled than when it is not. It should be noted that U_g for the controller case remains positive for CI > 2, i.e., as soon as the controller's effects start to be felt, while U_g goes to negative territory at CI = 12 and CI =24 for the non-controller case, when VM 2's workload has high peaks in workload intensity.

Figure 6.13 shows the variation of the CPU shares of VMs 1 and 2 during the 60-minute experiment. They both start with the same number of CPU shares each, i.e., 50%. As CPU resources need to be shifted between VMs, the controller changes the share allocation automatically. For example, at CI = 7, VM 1 sees a peak in the number of shares and VM 2 a valley, which coincides with the peaks and valleys seen

in the workload intensity of Fig. 6.9. The same happens at CI = 19.

Figures 6.14 and 6.15 show the variation of the response time for VMs 1 and 2, respectively, for the controller and non-controller cases, as well as the SLA (dashed line) for each case. For VM1, the response time stays below the SLA of 0.5 sec in both cases, controller and non-controller, except when the workload for VM 1 goes through its peak period. In these intervals, the response time under the controller case is significantly higher than when the controller is not used. The fact that VM 1 has a lower weight in the global utility function U_g causes the controller to tolerate temporary violations of SLA for VM 1. It should be noted that VM 1 received more shares during these peak periods. If that had not been done, the peaks in response time would have been higher. It should be noted also that in the non-controller case, VM 1 uses 50% of the CPU all the time while with the controller it never goes above 40%.

For the case of VM 2, Fig. 6.15 shows that as soon as the controller starts to operate (i.e., at CI = 2), the response time goes below its SLA of 0.030 sec and stays at that value throughout the entire experiment, except at CI = 24, when it goes only slightly above.The figure also shows that without the controller, this workload exceeds its SLA target by a significant margin at the peak periods for VM 2.

```
LevelList_0 ← a⃗_0;
CandidateList ← LevelList_0;
For i = 1 to d Do
    Begin
        LevelList_i ← ∅;
        For each a⃗ ∈ LevelList_{i−1} Do
            LevelList_i ← LevelList_i ∪ V(a⃗);
        LevelList_i ← Top (k, LevelList_i);
        CandidateList ← CandidateList ∪LevelList_i;
    End;
a⃗_opt ← max (CandidateList)
```

Figure 6.1: Controller Algorithm.

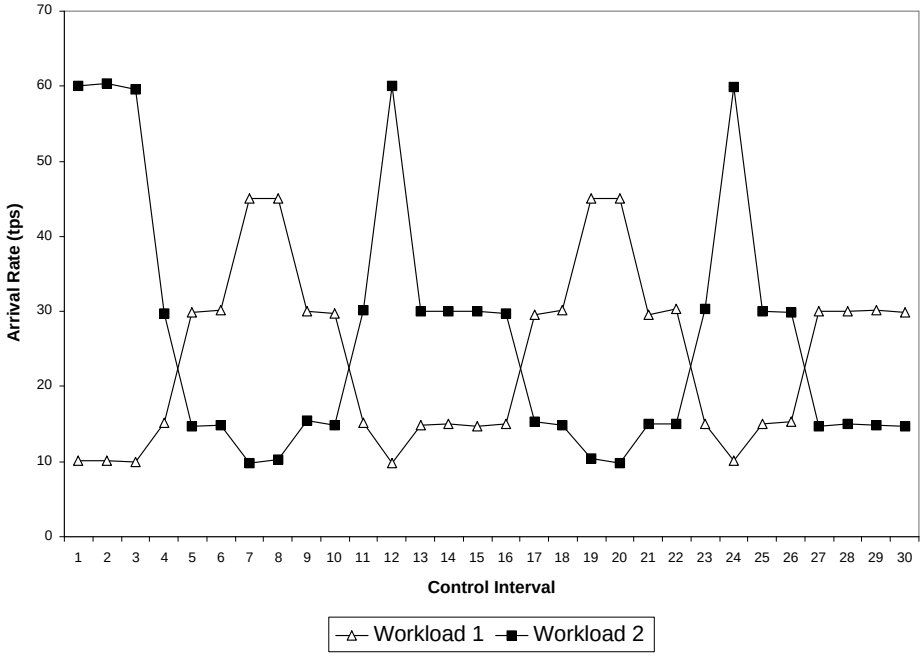


Figure 6.2: Variation of the workload intensity, in tps, for the two workloads as a function of time, in CIs, for the priority case.

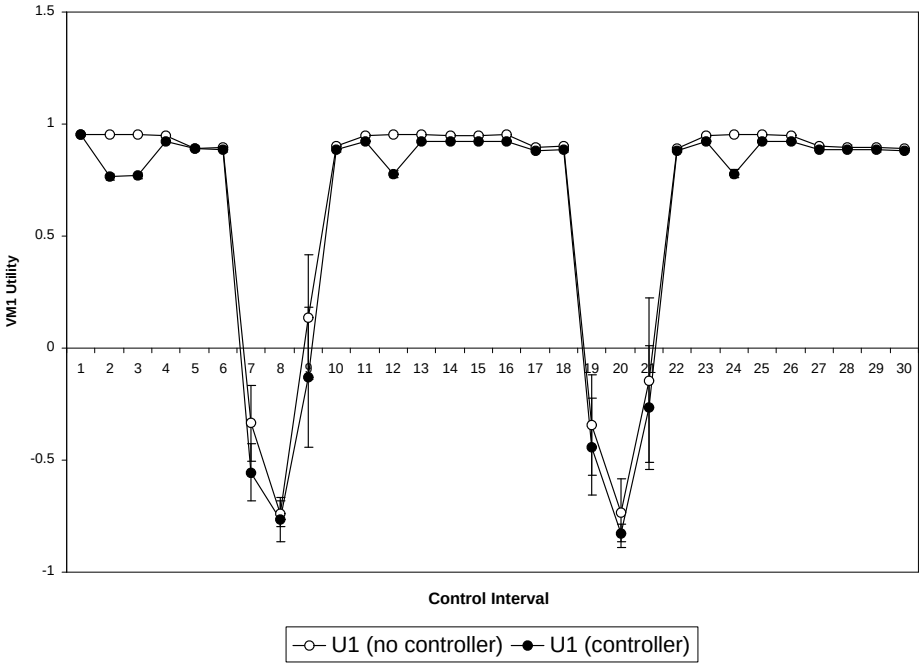


Figure 6.3: Variation of U_1 as a function of time, measured in CIs, for the priority case.

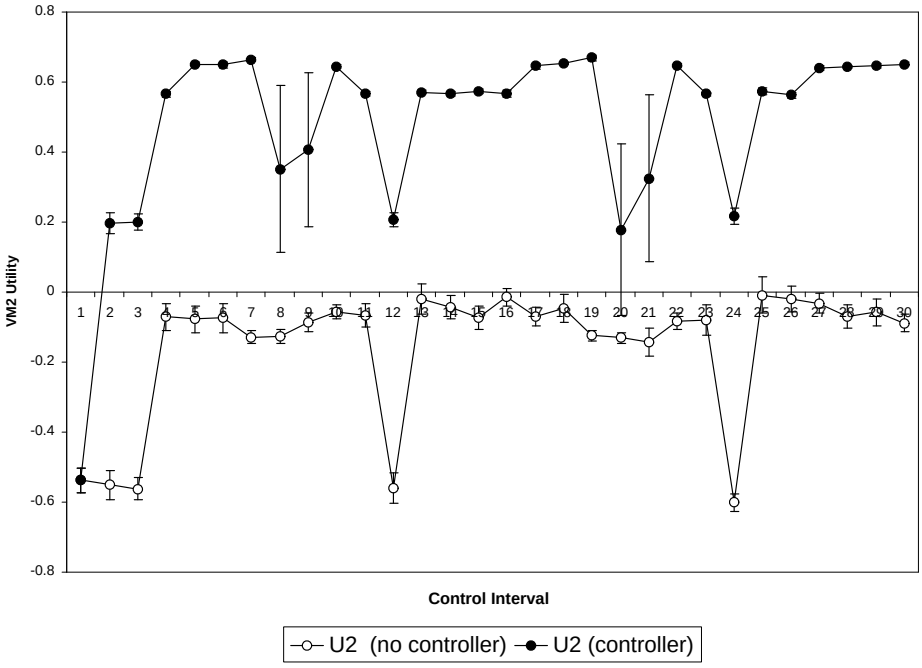


Figure 6.4: Variation of U_2 as a function of time, measured in CIs, for the priority case.

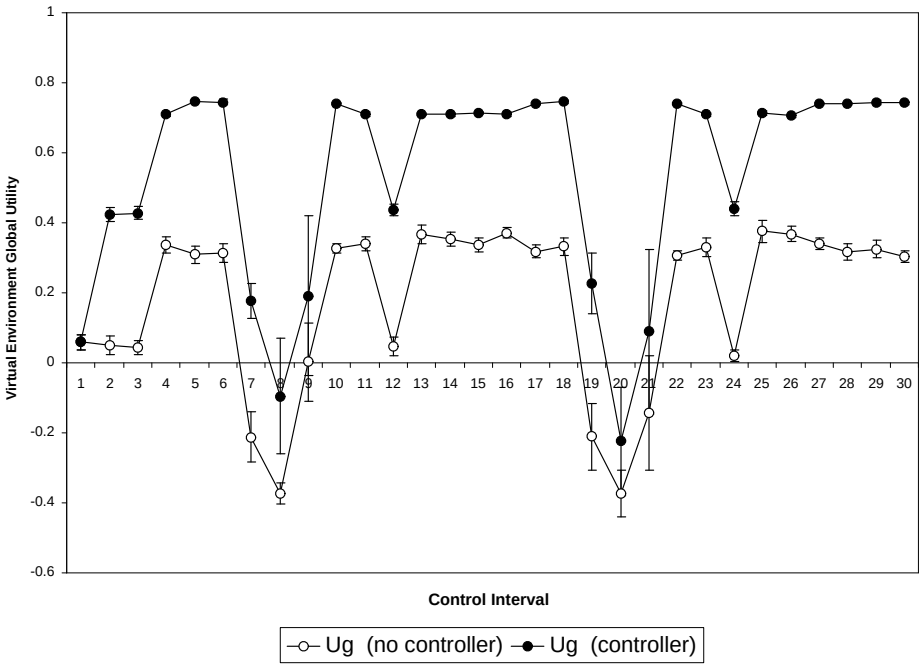


Figure 6.5: Variation of U_g as a function of time, measured in CIs, for the priority case.

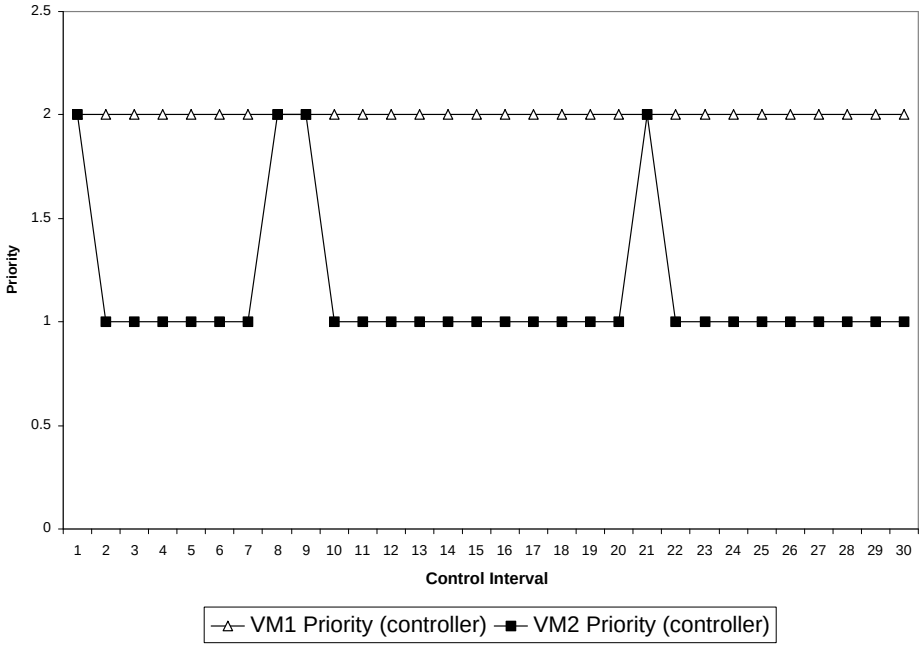


Figure 6.6: Variation of priorities of VMs 1 and 2 as a function of time, measured in CIs.

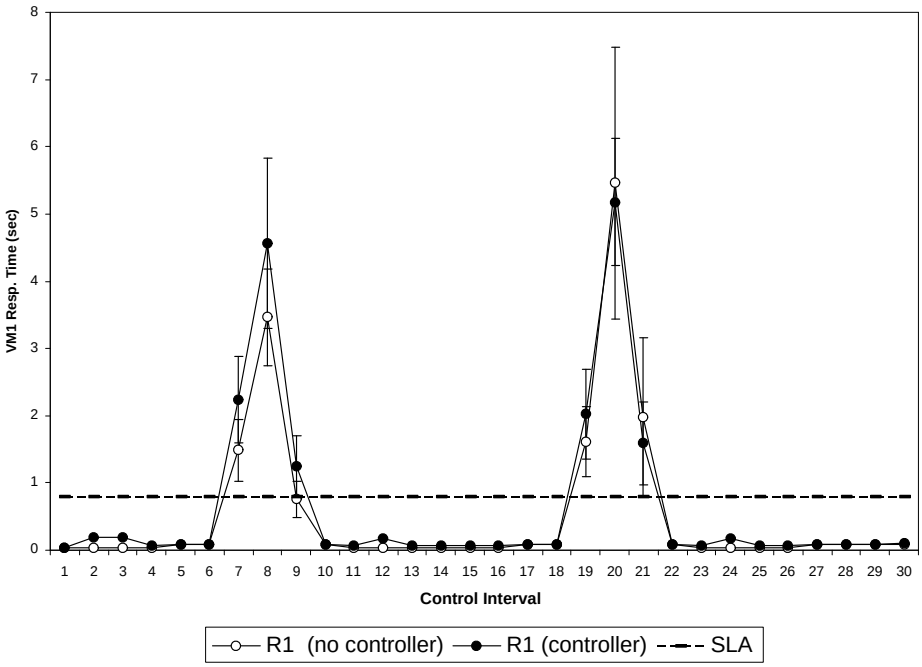


Figure 6.7: Variation of response time for VM 1 as a function of time, in CIs, for the priority case.

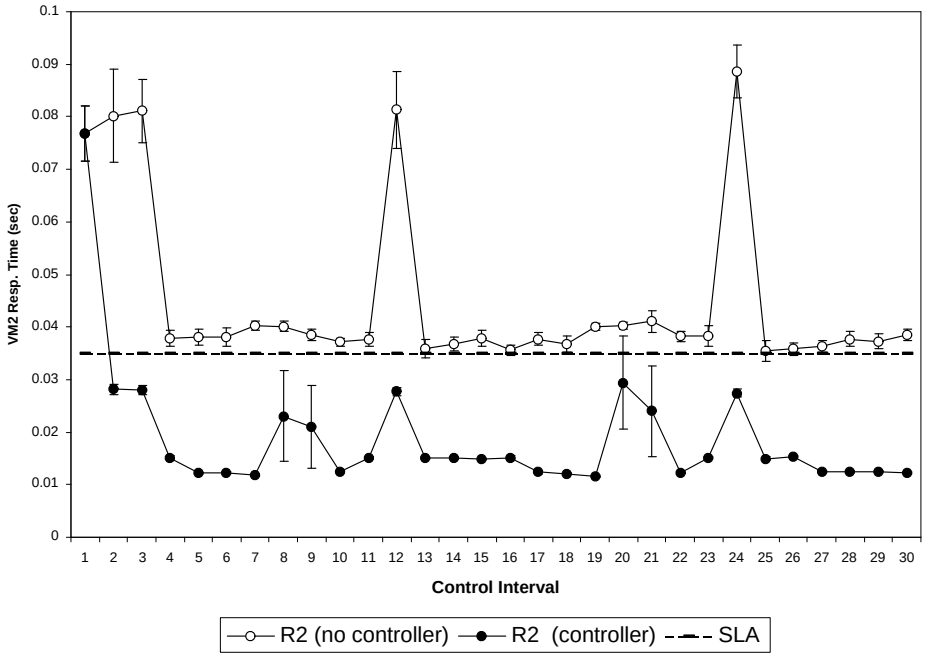


Figure 6.8: Variation of response time for VM 2 as a function of time, in CIs, for the priority case.

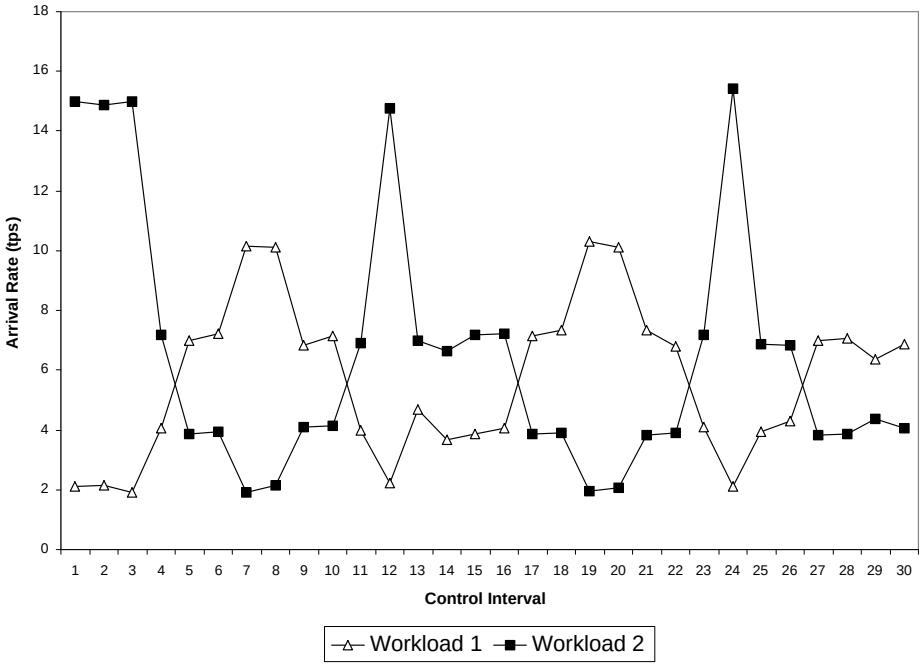


Figure 6.9: Variation of the workload intensity, in tps, for the two workloads as a function of time, in CIs, for the CPU share case.

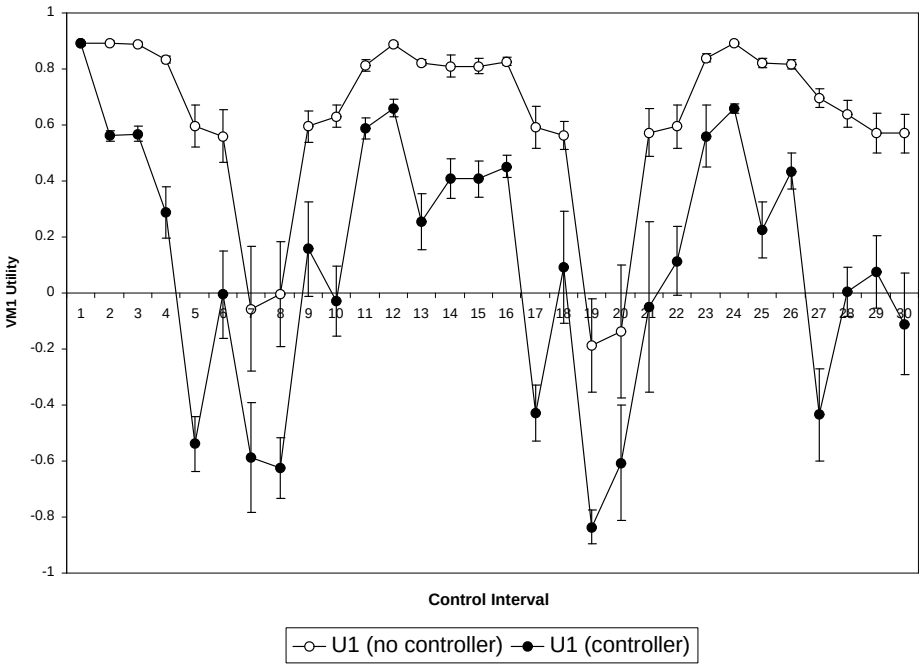


Figure 6.10: Variation of U_1 as a function of time, in CIs, for the CPU shares case.

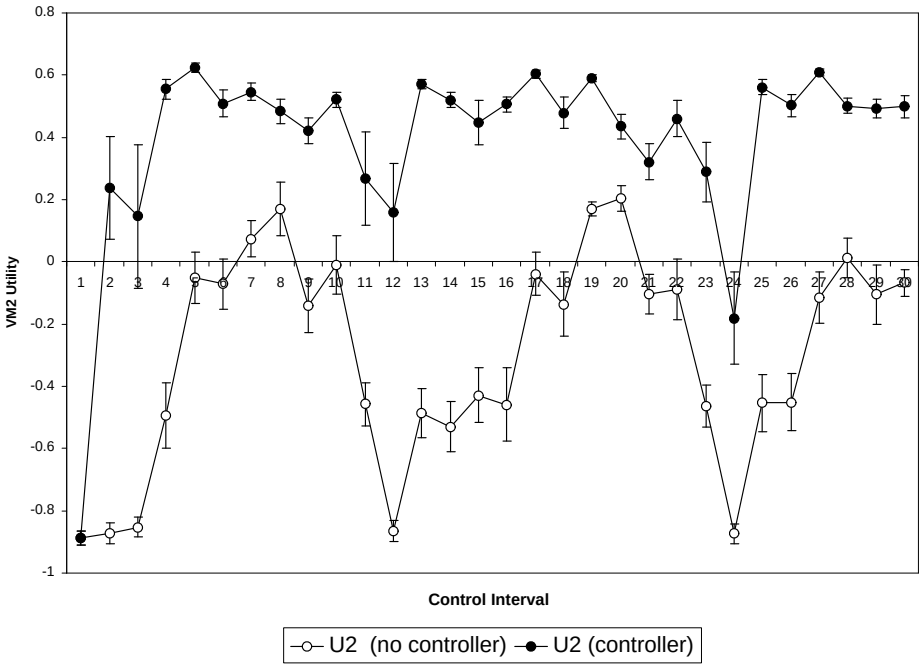


Figure 6.11: Variation of U_2 as a function of time, in CIs, for the CPU shares case.

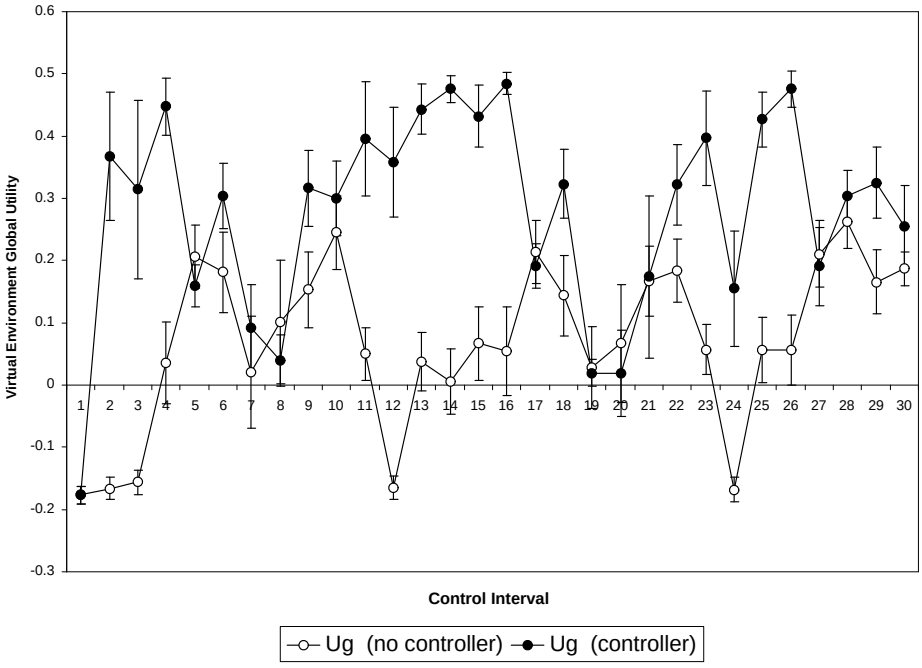


Figure 6.12: Variation of U_g as a function of time, in CIs, for the CPU shares case.

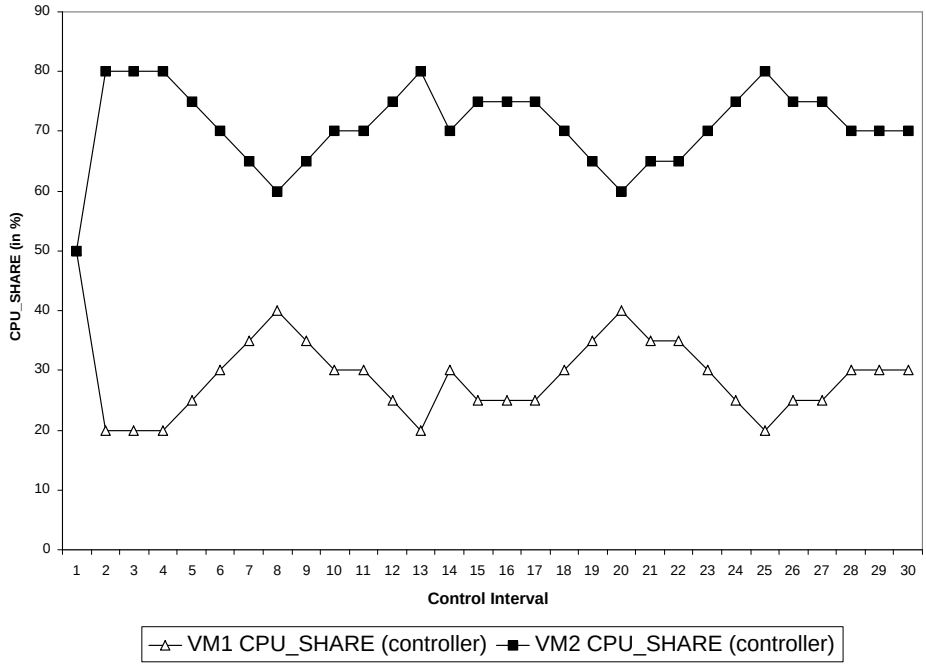


Figure 6.13: Variation of the CPU shares of VMs 1 and 2 as a function of time, in CIs.

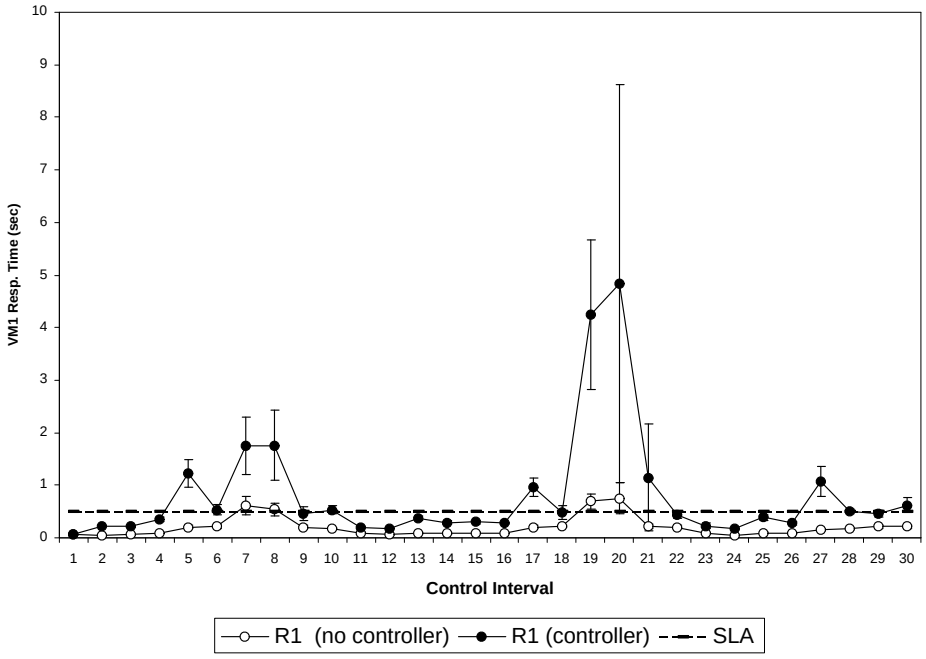


Figure 6.14: Variation of response time for VM 1 as a function of time, in CIs, for the CPU shares case.

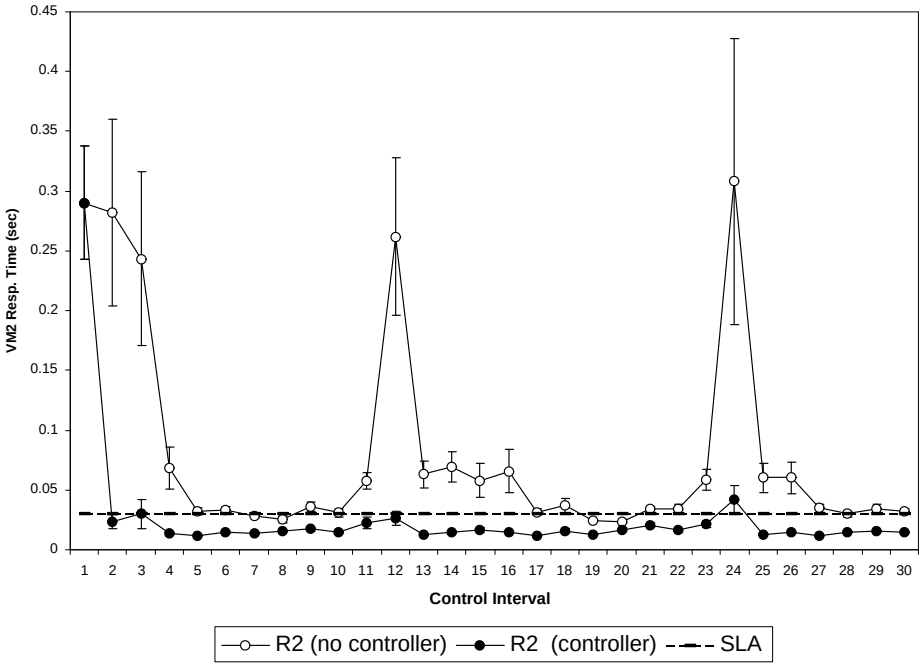


Figure 6.15: Variation of response time for VM 2 as a function of time, in CIs, for the CPU shares case.

Chapter 7: Conclusion and Future Work

7.1 Contributions

In this research we showed how autonomic systems can be empowered by the use of predictive analytic performance models. The idea revolves around the concept of augmenting autonomic systems by adding a controller engine and building mechanisms into the autonomic systems to allow them to receive control decisions from the controller. The controller's goal is to determine and communicate new configuration changes to the autonomic system on a regular basis in order to keep the system running at its peak performance.

We demonstrated how our proposed control approach can achieve this goal in an efficient manner. We presented and evaluated controller algorithms that are used to drive new configuration decisions for the case of three autonomic systems, namely an autonomic multithreaded server, an autonomic Internet data center, and an autonomic virtualized server. For each of these autonomic systems, the controller algorithm makes use of a combination of online analytic performance models for the system and a combinatorial search technique. The online analytic performance model guides the heuristic search technique in exploring the state of possible configuration vectors. A configuration vector is a vector whose elements denote either values of configuration parameters or resource allocation settings that can be changed dynamically. Online analytic performance models are helpful to the heuristic search technique in

the sense that they can predict the values of performance metrics that correspond to any configuration vector. The heuristic search method uses this knowledge to avoid exploring configuration vectors that are not very promising performance-wise. Therefore, the heuristic search technique reduces its exploration time and returns the best configuration vector at the end. The controller algorithm instructs, then, the autonomic system to implement the new configuration changes.

For the case of an autonomic multithreaded server we showed the effectiveness of our approach for the case of a simulated server and a real Apache Web server subject to a workload generated by the SURGE workload generator. We evaluated the impact of several control decisions including the choice of a particular heuristic serach technique, the use of workload forecasting, and the frequency of control. We also assessed the robustness of the system under highly variable workload conditions, the sensitivity of the system with respect to the service level agreements and the relative importance of the different performance metrics of interest. We extended our approach to the case of an autonomic multithreded server with multiple classes of customer requests.

For the case of autonomic Internet data centers, we showed how self-management can be achieved through the use of analytic performance models that scale very well with respect to the number of: application environments, resources, and classes of customer requests. We also contrasted our approach to a machine learning one. The work on the autonomic Internet data center provided us with a chance to deal with an additional design issue for the controller, namely the use of thin vs. fat local and global controllers. For the case of the simulated Internet data center, the global controller is fat while the local controllers are thin. For the case of the real Internet Data

center, on the other hand, the global controller is thin while the local controllers are fat. We also showed that our control approach is not only efficient at maintaining an overall higher performance for the entire Internet data center, but that the technique is robust enough to cope with situations where servers may fail. Our experimental results demonstrated that even in the presence of resource failures, the controller can make the right decisions of replacing the failed servers by operational ones from other application environments that are not that much in need of extra computing capacity.

For the case of an autonomic virtual server, we showed through simulation how our control approach can be used in an effective and efficient manner to drive the CPU allocation among the different hosted virtual machines. We considered both the cases of priority based and CPU share based allocations.

7.2 Future Research

The work on autonomic virtualized environments is still at a preliminary, but promising, stage. Future work on this topic should include implementing these techniques on an actual machine running a virtual machine monitor and several virtual machines with different workloads. The use of an open source virtual machine monitor such as Xen running on top of Linux could provide a very good experimental testbed for this kind of research.

The controller approach proposed in this research requires that a performance model be built for the system to be controlled. Moreover, such model should be detailed enough to capture the effect on performance of the parameters to be controlled. This endeavor requires modeling expertise, which is not necessarily easy to come by. As a future research, one should look for methods for automatically generating models

for a system. These methods would probably be driven by a library of existing and possible models to be used with data collected from the system that relates inputs to outputs. The "automatic model finder" would try to "fit"' a performance model to the data. This is clearly not an easy task, but clearly a very interesting topic for future research.

Bibliography

Bibliography

[1] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira, "Characterizing Reference Locality in the WWW," Proc. IEEE Conference on Parallel and Distributed Information Systems, Miami Beach, Florida, December 1996.

[2] M. Arlitt, R. Friedrich, and T. Jin, "Workload Characterization of a Web Proxy in a Cable Environment," *ACM Performance Evaluation Review*, 27 (2), Aug. 1999, pp. 25–36.

[3] B. Abraham, and J. Ledolter, "Statistical Methods for Forecasting," John Wiley & Sons, 1983.

[4] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch, "Hippodrome: running circles around system administration," *Conference on File and Storage Technologies (FAST'02)*, Monterey, CA, Jan. (2002)

[5] M. Arlitt and C. Williamson, "Web Server Workload Characterization: the Search for Invariants," *Proc. 1996 ACM Sigmetrics Conference on Measurement & Modeling of Computer Systems*, Philadelphia, PA, May 23-26, pp. 126–137.

[6] O. Babaoglu, M. Jelasity, A. Montresor, "Grassroots Approach to Self-Management in Large-Scale Distributed Systems," *In Proceedings of the EU-NSF Strategic Research Workshop on Unconventional Programming Paradigms, Mont Saint-Michel*, France, September 15–17, 2004.

[7] O. Babaoglu, M. Jelasity, A. Montresor, C. Fetzer, S. Leonardi, A. V. Moorsel, and M. V. Steen, eds., Self-Star Properties in Complex Information Systems, Lecture Notes in Computer Science, Vol. 3460, Springer Verlag, 2005.

[8] A. Bianco, M. G. Ben, E. Martinez, and V. Yohai, "Robust procedure for regression models with ARIMA errors," COMPSTAT'96 Proceedings of Computational Statistics, 27-38, Physica-Verlag, 1996.

[9] P. Barford, and M. Crovella, "Generating Representative Web Workloads for Network and Server Performance Evaluation," *Proc. 1998 ACM Sigmetrics*, Madison, Wisconsin, June 22-26, 1998.

[10] 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.

[11] M. N. Bennani and D. A. Menascé, "Resource Allocation for Autonomic Data Centers Using Analytic Performance Models," *Proc. IEEE Intl. Conf. Autonomic Computing (ICAC'05)*, Seattle, WA, June 13-16, 2005.

[12] M. Crovella, and A. Bestavros, "Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes," *IEEE/ACM Transactions on Networking*, 5(6), pp. 835–846, December 1997.

[13] J. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle, "Managing Energy and Server Resources in Hosting Centers," *18th Symp. Operating Systems Principles,* October 2001.

[14] 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)

[15] L. Cherkasova and P. Phaal, "Session Based Admission Control: A Mechanism for Improving the Performance of an Overloaded Web Server," HPL-98-119, HP Labs Technical Reports, 1998.

[16] CSIM, Mesquite Software, http://www.mesquite.com/

[17] L. Chang, G. C. Tiao, and C. Chen, "Estimation of series in the presence of outliers," Technometrics, 30, No.2, 193-204, 1988.

[18] 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.

[19] 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.

[20] D. L. Eager, D. J. Sorin, and M. K. Vernon, "AMVA Techniques for High Service Time Variability," *Proc. ACM SIGMETRICS 2000 Int'l. Conf. on Measurement and Modeling of Computer Systems*, Santa Clara, CA, June 2000.

[21] D. Garlan, S. Cheng, and B. Schmerl: Increasing System Dependability through Architecture-based Self-repair. Architecting Dependable Systems, R. de Lemos, C. Gacek, A. Romanovsky (eds.), Springer-Verlag, (2003)

[22] W. Gentzsch, K. Iwano, D. Johnston-Watt, M. A. Minhas, M. Yousif, "Self-Adaptable Autonomic Computing Systems: An Industry View," *In Proceedings of sixteenth International Workshop on Database and Expert Systems Applications,* 2005.

[23] M. Goldszmidt, D. Palma, and B. Sabata, "On the Quantification of e-Business Capacity," *Proc. 2001 ACM Conference on E-commerce,* Tampa, FL, October 14-17, 2001, pp. 235–244.

[24] Hewlett-Packard. Infrastructure and Management Solutions for the Adaptive Enterprise.
http://www.hp.com/products1/promos/adaptive_enterprise/pdfs/vision_for_ae.pdf

[25] Hewlett-Packard. VSE Management Software: Quick Start Guide.
http://docs.hp.com/en/T2786-90001/T2786-90001.pdf

[26] Hewlett-Packard. HP-UX Workload Manager Overview.
http://www.hp.com/products1/unix/operating/docs/wlm.overview.pdf

[27] IBM. Autonomic Computing: IBM's Perspective on the State of Information Technology.
http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf

[28] IMSL, Visual Numerics, http://www.vni.com/

[29] Intel. Autonomic Platform Research (APR).
http://www.intel.com/cd/ids/developer/asmona/eng/192589.htm

[30] R. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*, John Wiley, NY, 1991.

[31] M. Jelasity, A. Montresor, and O. Babaoglu, "A Modular Paradigm for Building Self-Organizing Peer-to-Peer Applications,". *In Post-Proceedings of ESOP03: International Workshop on Engineering Self-Organising Applications,* Lecture Notes in Computer Science, vol. 2977, Springer-Verlag, Berlin 2004.

[32] J. Kephart and D. Chess, "The Vision of Autonomic Computing," IEEE Computer, Vol. 36, No. 1, January 2003, p. 41-50

[33] J. Kephart, "Research Challenges of Autonomic Computing," Proc. of the 27th international conference on Software engineering, St. Louis, MO, 2005

[34] A. Kermarrec, "Self-clustering in Peer-to-Peer overlays,". *In International Workshop on Self-* Properties in Complex Information Systems,* Bertinoro, Italy, February (2004) 89–92

[35] L. Kleinrock, *Queuing Systems, Volume I: Theory*, Wiley-Interscience, NY, 1975.

[36] D. M. Levine, P. P. Ramsey, R. K. Smidt, *Applied Statistics for Engineers and Scientists: Using Microsoft Excel and MINITAB*, Prentice Hall, Upper Saddle River, 2001.

[37] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative System Performance: Computer System Analysis Using Queueing Network Models,* Prentice Hall, Englewood Cliffs, NJ, 1984.

[38] D. A. Menascé, and V. A. F. Almeida, *Web Performance: Metrics, Models, and Methods,* Prentice Hall, Upper Saddle River, NJ, 1998.

[39] D. A. Menascé and V. A. F. Almeida, *Scaling for E-business: technologies, models, performance, and capacity planning,* Prentice Hall, Upper Saddle River, NJ, 2000.

[40] D. A. Menascé and V. A. F. Almeida, *Capacity Planning for Web Services: metrics, models, and methods,* Prentice Hall, Upper Saddle River, NJ, 2002.

[41] D. A. Menascé, V. A. F. Almeida, and L. W. Dowdy, *Capacity Planning and Performance Modeling: from mainframes to client-server systems,* Prentice Hall, 1994.

[42] D. A. Menascé, V. A. F. Almeida, and L. W. Dowdy, *Performance by Design: Computer Capacity Planning by Example*, Prentice Hall, Upper Saddle River, NJ, 2004.

[43] D. A. Menascé, V. A. F. Almeida, R. C. Fonseca, and M. A. Mendes, "A Methodology for Workload Characterization for E-commerce Servers," *Proc. 1999 ACM Conference in Electronic Commerce*, Denver, CO, Nov. 1999.

[44] D. A. Menascé, V. Almeida, R. Fonseca, and M. A. Mendes, "Business-oriented Resource Management Policies for E-commerce Servers," *Performance Evaluation*, September 2000.

[45] D. A. Menascé, V. A. F. Almeida, R. Riedi, F. Pelegrinelli, R. Fonseca, and W. Meira Jr., "In Search of Invariants in E-Commerce Workloads," *Proc. 2000 ACM Conference in Electronic Commerce*, Minneapolis, MN, Oct. 17-20, 2000.

[46] 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.

[47] D. A. Menascé, M. N. Bennani and H. Ruan "On the Use of Online Analytic Performance Models in Self-Managing and Self-Organizing Computer Systems," in the book Self-Star Properties in Complex Information Systems, O. Babaoglu, M. Jelasity, A. Montresor, C. Fetzer , S. Leonardi, A. van Moorsel, and M. van Steen, eds., Lecture Notes in Computer Science, Vol. 3460, Springer Verlag, 2005.

[48] 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.

[49] D. A. Menascé, "Virtualization: Concepts, Applications, and Performance Modeling," Proc. 2005 Computer Measurement Group Conf., Orlando, FL, Dec. 4-9, 2005.

[50] D. A. Menascé, and E. Casalicchio, "A Framework for Resource Allocation in Grid Computing," *Proc. 12th Annual Meeting of the IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS),* Volendam, The Netherlands, October 5–7, 2004.

[51] D. A. Menascé, "Automatic QoS Control," IEEE Internet Computing, Jan./Febr. 2003, Vol. 7, No. 1.

[52] D. A. Menascé, "Security Performance," IEEE Internet Computing (2003) May/June, Vol. 7, No. 3, 84–87

[53] D. A. Menascé, "Performance and Availability of Internet Data Centers," IEEE Internet Computing (2004) May/June, Vol. 8, No. 3, 94–96

[54] D. A. Menascé, "Mapping Service Level Agreements in Distributed Applications," IEEE Internet Computing (2004) September/October , Vol. 8, No. 5, 100–102

[55] D. A. Menascé and M. N. Bennani, "Dynamic Server Allocation for Autonomic Service Centers in the Presence of Failures", handbook on Autonomic Computing, "Autonomic Computing: Concepts, Infrastructure, and Applications", eds. Manish Parashar and Salim Hariri, CRC Press 2006.

[56] D. A. Menascé and M. N. Bennani, "Autonomic Virtualized Environments," *Proc. IEEE International Conference on Autonomic and Autonomous Systems (ICAS'06)*, Silicon Valley, CA, July 19-21, 2006.

[57] Microsoft. Microsoft Dynamic Systems Initiative Overview. http://www.microsoft.com/windowsserversystem/dsi/dsiwp.mspx

[58] D. A. Menascé, H. Ruan, and H. Gomaa, "A Framework for QoS-Aware Software Components," *Proc. ACM 2004 Workshop on Software and Performance*, San Francisco, CA, January 14–16, (2004)

[59] D. Martin and V. Yohai, "Data Mining for Unusual Movements in Temporal Data," Proceedings of the KDD 2001 Workshop on Temporal Data Mining, August, San Francisco.

[60] Oracle. Oracle Automatic Workload Repository (AWR). http://www.oracle-base.com/articles/10g/AutomaticWorkloadRepository10g.php

[61] Peakstone Corporation, www.peakstone.com.

[62] J. Pitkow, "Summary of WWW characterizations," *World Wide Web*, No. 2, 1999.

[63] V. J. Rayward-Smith, I. H. Osman, and C. R. Reeves, eds, Modern Heuristic Search Methods, John Wiley & Sons, Dec. 1996.

[64] F. Schintke, T. Schutt, and A. Reinefeld, "A Framework for Self-Optimizing Grids Using P2P Components," Intl. Workshop on Autonomic Computing Systems, Sep. (2003)

[65] S-PLUS, Insightful, http://www.insightful.com/

[66] M. S. Squillante, D. D. Yao, and L. Zhang "Internet Traffic: Periodicity, Tail Behavior and Performance Implications", ed. E. Gelenbe, "System Performance Evaluation: Methodologies and Applications", CRC Press 1999.

[67] G. Tesauro, R. Das, W. E. Walsh, and J. O. Kephart, "Utility-Function-Driven Resource Allocation in Autonomic Systems," *Proc. IEEE Intl. Conf. Autonomic Computing (ICAC'05)*, Seattle, WA, June 13-16, 2005.

[68] G. Tesauro, "Online Resource Allocation Using Decompositional Reinforcement Learning," *Proc. AAAI-05*, Pittsburgh, PA, July 9-13, 2005.

[69] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani, "A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation," *Proc. IEEE Intl. Conf. Autonomic Computing (ICAC'06)*, Dublin, Ireland, June 13-16, 2006.

[70] WebSphere Benchmark SampleS-PLUS, IBM, http://www-306.ibm.com/software/webservers/appserv/benchmark3.html

[71] 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.

[72] WebQoS, Hewlett-Packard, http://www.hp.com/products1/webqos/

[73] WebSphere Extended Deployment, IBM, http://www.ibm.com/software/webservers/appserv/extend/

[74] R. Wickremisinghe, J. Vitter, and J. Chase, "Distributed Computing with Load-Managed Active Storage," IEEE Int. Symp. High Performance Distr. Computing, July (2002)

[75] P. H. Winston, "Artificial Intelligence," 3rd edition, Addison Wesley, New York, 1993.