RC23646 (W0506-124) June 29, 2005
Computer Science

IBM Research Report

A Control Theory Foundation for Self-Managing Computing Systems

Yixin Diao, Joseph L. Hellerstein, Sujay Parekh
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Rean Griffith, Gail Kaiser, Dan Phung
Computer Science Department
Columbia University
New York, NY

A Control Theory Foundation for Self-Managing Computing Systems

Yixin Diao, Joseph L. Hellerstein, and Sujay Parekh
IBM Thomas J. Watson Research Center
Hawthorne, New York, USA
{diao, hellers, sujay}@us.ibm.com

Rean Griffith, Gail Kaiser and Dan Phung
Computer Science Department
Columbia University, New York, NY
{rg2023, kaiser, phung}@cs.columbia.edu

May 30, 2005

Abstract

The high cost of operating large computing installations has motivated a broad interest in reducing the need for human intervention by making systems self-managing. This paper explores the extent to which control theory can provide an architectural and analytic foundation for building self-managing systems. Using the IBM autonomic computing architecture as a starting point, we show the correspondence between the elements of autonomic systems and those in control systems. For example, the sensors and effectors of the autonomic architecture provide the measured outputs and control inputs in control systems. The benefit of making this connection is that control theory provides a rich set of methodologies for building automated self-diagnosis and self-repair systems with properties such as stability, short settling times, and accurate regulation. This said, there remain considerable challenges in applying control theory to computing systems, such as developing effective resource models, handling sensor delays, and addressing lead times in effector actions. To illustrate these challenges, we present a case study in which control theory has been used in an IBM database management product. We believe that addressing these challenges requires a broader engagement of the research community. To this end, we propose a deployable testbed for autonomic computing (DTAC) that we believe will reduce the barriers to addressing key research problems in autonomic computing. The initial DTAC architecture is described along with several problems that it can be used to investigate.

1 Introduction

The high cost of ownership of computing systems has resulted in a number of industry initiatives to reduce the burden of operations and management. Examples include IBM's Autonomic Computing, HP's Adaptive Infrastructure, and Microsoft's Dynamic Systems Initiative. All of these efforts seek to reduce operations costs by increased automation, ideally to have systems be self-managing without any human intervention (since operator error has been identified as a major source of system failures [1]). While the concept of automated operations has existed for two decades (e.g., [2]) as a way to adapt to changing workloads, failures and (more recently) attacks, the scope of automation remains limited. We believe this is in part due to the absence of a fundamental understanding of how automated actions affect system behavior, especially system stability. Other disciplines such as mechanical, electrical, and aeronautical engineering make use of control theory to design feedback systems. This paper uses control theory as a way to identify a number of requirements for and challenges in building self-managing systems.

The IBM Autonomic Computing Architecture [3] provides a framework in which to build self-managing systems. We use this architecture since it is broadly consistent with other approaches that have been developed (e.g., [4]). Figure 1 depicts the components and key interactions for a single autonomic manager and a single resource. The resource (sometimes

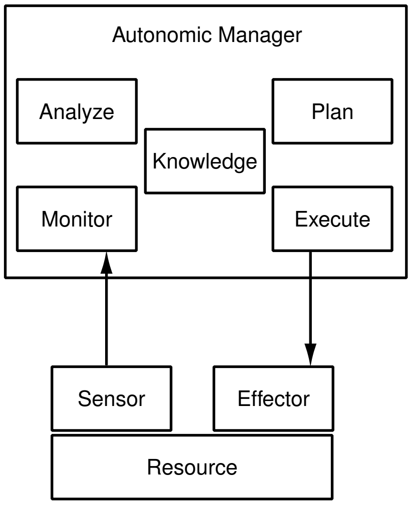


Figure 1: *Architecture for Autonomic Computing.*

called a managed resource) is what is being made more self-managing. This could be a single system (or even an application within a system), or it may be a collection of many logically related systems. Sensors provide a way to obtain measurement data from resources, and effectors provide a means to change the behavior of the resource. Autonomic managers read sensor data and manipulate effectors to make resources more self-managing. The autonomic manager contains components for monitoring, analysis, planning, and execution. Common to all of these is knowledge of the computing environment, service level agreements, and other related considerations. The monitoring component filters and correlates sensor data. The analysis component processes these refined data to do forecasting and problem determination, among other activities. Planning constructs workflows that specify a partial order of actions to accomplish a goal specified by the analysis component. The execute component controls the execution of such workflows and provides coordination if there are multiple concurrent workflows. (The term "execute" may be broadened to "enactment" to include manual actions as well.)

In essence, the autonomic computing architecture provides a blue print for developing feedback control loops for self-managing systems. This observation suggests that control theory might provide guidance as to the structure of and requirements for autonomic managers.

Many researchers have applied control theory to computing systems. In data networks, there has been considerable interest in applying control theory to problems of flow control, such as [5] who develops the concept of a Rate Allocating Server that regulates the flow of packets through queues. Others have applied control theory to short-term rate variations in TCP (e.g., [6]) and some have consider stochastic control [7]. More recently, there have been detailed models of TCP developed in continuous time (using fluid flow approximations) that have produced interesting insights into the operation of buffer management schemes in routers (see [8], [9]). More recently, control theory has been applied to middleware to provide service differentiation and regulation of resource utilizations as well as optimization of service level objectives. Examples of service differentiation include enforcing relative delays [10], preferential caching of data [11], and modifying application codes to insert effectors (e.g., [12]). Examples of regulating resource utilizations include a mixture of queueing and control theory used to regulate the Apache HTTP Server [13], regulation of the IBM Lotus Domino Server [14], and multiple-input, multiple-output control of the Apache HTTP Server (e.g., simultaneous regulation of CPU and memory resources) [15]. Examples of optimizing service level objectives include minimizing response times of the Apache Web Server [16] and balancing the load to optimize database memory management [17].

The foregoing illustrates the value of using control theory to construct self-managing systems as first described in [18]. Section 2 seeks to educate systems oriented computer science researchers and practitioners on the concepts and techniques needed to apply control theory to computing systems. Section 3 describes how control theory can aid in building self-managing systems, and identifies the challenges in doing so. Section 4 presents a case study of applying control theory to an IBM database management product. Section 5 proposes a deployable testbed for autonomic computing that is intended to foster research that addresses

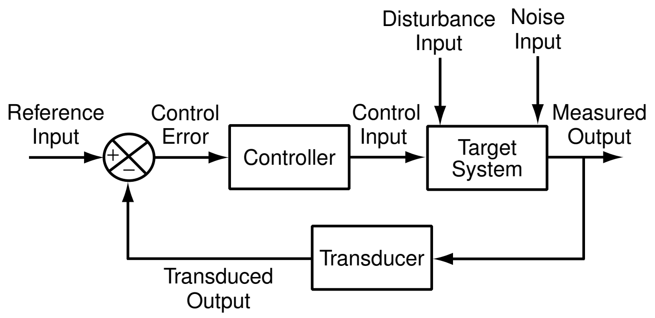


Figure 2: *Block diagram of a feedback control system. The reference input is the desired value of the system's measured output. The controller adjusts the setting of control input to the target system so that its measured output is equal to the reference input. The transducer represents effects such as units conversions and delays.*

the challenges identified. Our conclusions are contained in Section 6.

2 Control Theoretic Framework

This section relates control theory to self-managing systems.

2.1 Components of a Control System

Over the last sixty years, control theory has developed a fairly simple reference architecture. This architecture is about manipulating a target system to achieve a desired objective. The component that manipulates the target system is the controller. In terms of Figure 1, the target system is a resource, the controller is an autonomic manager, and the objective is part of the policy knowledge.

The essential elements of feedback control system are depicted in Figure 2. These elements are:

- control error, which is the difference between the reference input and the measured output.

- control input, which is a parameter that affects the behavior of the target system and can be adjusted dynamically (such as the `MaxClients` parameter in the Apache HTTP Server).
- controller, which determines the setting of the control input needed to achieve the reference input. The controller computes values of the control input based on current and past values of control error.
- disturbance input, which is any change that affects the way in which the control input influences the measured output (e.g., running a virus scan or a backup).
- measured output, which is a measurable characteristic of the target system such as CPU utilization and response time.
- noise input, which is any affect that changes the measured output produced by the target system. This is also called sensor noise or measurement noise.
- reference input, which is the desired value of the measured output (or transformations of them), such as CPU utilization should be 66%. Sometimes, the reference input is referred to as desired output or the setpoint.
- target system, which is the computing system to be controlled.
- transducer, which transforms the measured output so that it can be compared with the reference input (e.g., smoothing stochastics of the output).

The foregoing is best understood in the context of a specific system. Consider a cluster of Apache Web Servers. The Administrator may want these systems to run at no greater than 66% utilization so that if any one of them fails, the other two can immediately absorb the entire load. Here, the measured output is CPU utilization. The control input is the maximum number of connections that the server permits as specified by the `MaxClients` parameter. This parameter can be manipulated to adjust CPU utilization. Examples of disturbances are changes in arrival rates and shifts in the type of requests (e.g., from static to dynamic pages).

While the autonomic computing and control systems architectures are very similar, there are some important differences, mostly in emphasis. Autonomic computing focuses on the specification and construction of components that interoperate well for management tasks. For example, the autonomic computing architecture focuses sensors and effectors since there are specific interfaces that must be developed.

In contrast, the emphasis in control theory is on analyzing and/or developing components and algorithms such that the resulting system achieves the control objectives. For example, control theory provides design techniques for determining the values of parameters in commonly used control algorithms so that the resulting control system is stable and settles quickly in response to disturbances.

2.2 Objectives and Properties of Control Systems

Controllers are designed for some intended purpose. We refer to this purpose as the control objective. The most common objectives are:

- regulatory control: Ensure that the measured output is equal to (or near) the reference input. For example, the utilization of a web server should be maintained at 66%. The focus here is on changes to the reference input such as changing the target utilization from 66% to 75% if a fourth server becomes available. Another example is service differentiation.
- disturbance rejection: Ensure that disturbances acting on the system do not significantly affect the measured output. For example, when a backup or virus scan is run on a web server, the overall utilization of the system is maintained at 66%. This differs

from regulator control in that we focus on changes to the disturbance input, not to the reference input.

- optimization: Obtain the “best” value of the measured output, such as optimizing the setting of `MaxClients` in the Apache HTTP Server so as to minimizes response times.

There are several properties of feedback control systems that should be considered when comparing controllers for computing systems. Our choice of metrics is drawn from experience with the commercial information technology systems. Other properties may be of interest in different settings. For example, [19] discuss properties of interest for control of real-time systems.

Below, we motivate and present the main ideas of the properties considered.

- A system is said to be *stable* if for any bounded input, the output is also bounded. Stability is typically the first property considered in designing control systems since unstable systems cannot be used for mission critical work.

- The control system is *accurate* if the measured output converges (or becomes sufficiently close) to the reference input. Accurate systems are essential to ensuring that control objectives are met, such as differentiating between gold and silver classes of service and ensuring that throughput is maximized without exceeding response time constraints. Typically, we do not quantify accuracy. Rather, we measure inaccuracy. For a system in steady state, its inaccuracy, or **steady state error** is the steady state value of the control error.

- The system has *short settling times* if it converges quickly to its steady state value. Short settling times are particularly important for disturbance rejection in the presence of time-varying workloads so that convergence is obtained before the workload changes.

- The system should achieve its objectives in a manner that *does not overshoot.* Consider a system in which the objective is to maximize throughput subject to the constraint that response time is less than one second, which is often achieved by a regulator that keeps response times at their upper limit so that throughput is maximized. Suppose that incoming requests change so that they are less CPU intensive and hence response times decrease to 0.5 seconds. Then, by avoiding overshoot, we mean that as the controller changes the control input that causes throughput to increase (and hence response time to increase), response times should not exceed one second. Overshoot is particularly important if exceeding a limit has significant consequences, such as a buffer overflow.

Much of our application of control theory is based on the properties of stability, accuracy, settling time, and overshoot. We refer to these as the **SASO properties**.

To elaborate on the SASO properties, we consider what constitutes a stable system. For computing systems, we want the output of feedback control to converge, although it may not be constant due to the stochastic nature of the system. To refine this further, computing systems have operating regions (i.e., combinations of workloads and configuration settings) in which they perform acceptably and other operating regions in which they do not. Thus, in general, we refer to the stability of a system within an operating region. Clearly, if a system is not stable, its utility is severely limited. In particular, the system's response times will be large and highly variable, a situation that can make the system unusable.

Figure 3 displays an instability in an the Apache HTTP Server that employs an improperly designed controller. The horizontal axis is time, and the vertical axis is CPU utilization (which ranges between 0 and 1). The solid line is the reference input for CPU utilization, and the line with markers is the measured value. During the first 300 seconds, the system operates in open loop. When the controller is turned on, a reference input of 0.5 is used.

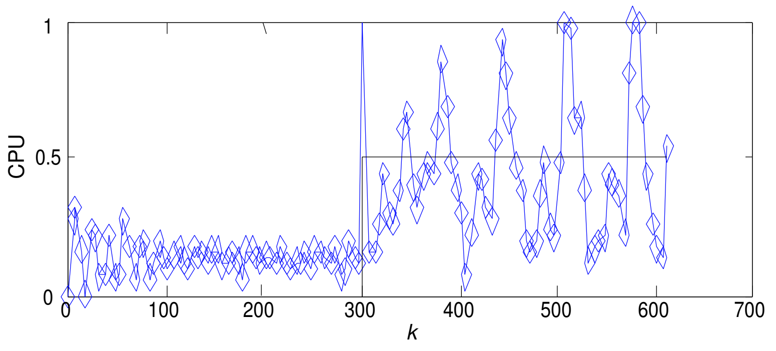


Figure 3: *Example of an unstable feedback control system for the Apache HTTP Server. The instability results from having an improperly designed controller.*

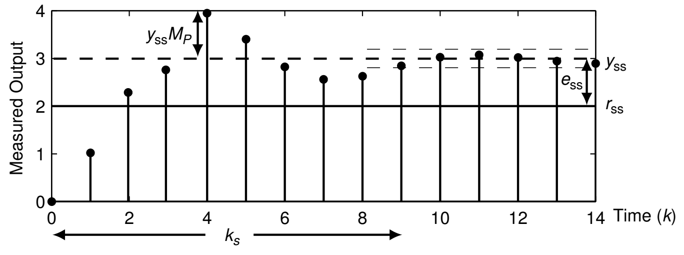


Figure 4: *Response of a stable system to a step change in the reference input. At time* 0*, the reference input changes from* 0 *to* 2*. The system reaches steady state when its output always lie between the light weight dashed lines. Depicted are the steady state error* (e_{ss})*, settling time* (k_s)*, and maximum overshoot* (M_{P})*.*

At this point, the system begins to oscillate and the amplitude of the oscillations increases. This is a result of a controller design that overreacts to the stochastics in the CPU utilization measurement. Note that the amplitude of the oscillations is constrained by the range of the CPU utilization metric.

If the feedback system is stable, then it makes sense to consider the remaining SASO properties—accuracy, settling time, and overshoot. The vertical lines in Figure 4 plot the measured output of a stable feedback system. Initially, the (normalized) reference input is 0. At time 0, the reference input is changed to its steady value $r_{ss} = 2$. The system responds and its measured output eventually converges to $y_{ss} = 3$, as indicated by the heavy dashed line. The steady state error e_{ss} is -1, where $e_{ss} = r_{ss} - y_{ss}$. The settling time of the system k_s is the time from the change in input to when the measured output is sufficiently close to its new steady state value (as indicated by the light dashed lines). In the figure, $k_s = 9$. The maximum overshoot M_P is the (normalized) maximum amount by which the measured output exceeds its steady state value. In the figure, the maximum value of the output is 3.95 and so $(1 + M_P)y_{ss} = 3.95$, or $M_P = 0.32$.

The properties of feedback systems are used in two ways. The first relates to the analysis. Here, we are interested in determining if the system is stable as well as measuring and/or estimating its steady state error, settling time, and maximum overshoot. The second is in the design of feedback systems. Here, the properties are design goals. That is, we construct the feedback system to have the desired values of steady state error, settling times, and maximum overshoot. More details on applying control theory to computing systems can be found in [20].

2.3 Control Analysis and Design

This subsection uses a running example to outline an approach to control analysis and design for self-managing systems with SASO properties.

We consider the IBM Lotus Domino Server. To ensure efficient and reliable operation, Administrators of this system often regulate the number of remote procedure calls (RPCs) in the server, a quantity that we denote by RIS. RIS roughly corresponds to the number of *active users* (those with requests outstanding at the server). Regulation is accomplished by using the `MaxUsers` tuning parameter that controls the number of *connected users*. The correspondence between `MaxUsers` and RIS changes over time, which means that `MaxUsers` must be updated almost continuously to achieve the control objective. Clearly, it is desirable to have a controller that automatically determines the value of `MaxUsers` based on the objective for RIS.

Our starting point is to model how `MaxUsers` affects RIS. The input to this model is `MaxUsers`, and the output is RIS. We use $u(k)$ to denote the k-th value of the former and $y(k)$ to denote the k-th value of the latter. (Actually, $u(k)$ and $y(k)$ are offsets from a desired operating point.) A standard workload was applied to a IBM Lotus Domino Server running product level software in order to obtain training and test data. In all cases, values are averaged over a one minute interval. We construct the following simple autoregressive model using least squares regression:

$$y(k+1) = 0.43y(k) + 0.47u(k) \tag{1}$$

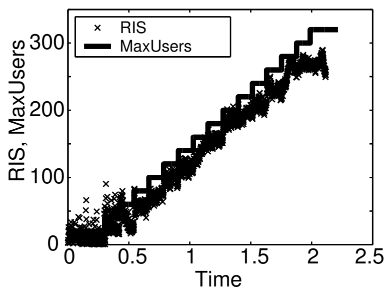
Figure 5(a) displays the values of $u(k)$ (solid line) and the corresponding $y(k)$ ("x"s). We see that the model fits these data quite well.

To better facilitate control analysis, Equation (1) is put into the form of a transfer function, which is a Z-transform representation of how `MaxUsers` affects RIS. Z-transforms provide a compact representation for time varying functions, where z represents a time shift operation. The transfer function of Equation (1) is

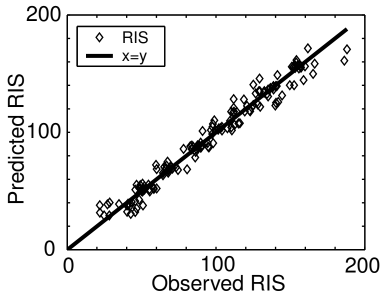
$$\frac{0.47}{z - 0.43}. \tag{2}$$

The transfer function of a system tells us about its steady state output and its settling times. We do this by computing the **steady state gain** of the transfer function, which is the value of the transfer function when $z = 1$. For example, Equation (2), the steady state gain is 0.82. This means that RIS will be (0.82)`MaxUsers` at steady state.

The **poles** of the transfer function provide a way to estimate the settling time of the system. The poles are the values of z for which the denominator is 0. For example, in Equation (2), there is one pole, which is 0.43. The effect of this pole on settling time is clear if we solve the recurrence in Equation (1). The result has the factors $0.43^{k+1}, 0.43^{k}, \cdots$. Thus, if the absolute value of the pole is greater than one, the system is unstable. And the closer the pole is to 0, the shorter the settling time. A pole that is negative (or imaginary) indicates an oscillatory response. In general, for a transfer function whose largest pole is p, its settling time is approximately $\frac{-4}{log|p|}$ [20]. Thus, the settling time for Equation (2) is approximately 5 time units.



(a) Data used in system identification



(b) Model evaluation

Figure 5: *Data and model evaluation for the IBM Lotus Domino Server.*

Figure 6 contains a block diagram of a control system that automatically adjusts `MaxUsers` to regulate the measured value of RIS. Note that the target system consists of two blocks—the Notes Server and the Notes Sensor. These blocks correspond to a resource and its sensor in the autonomic computing architecture. The Notes Sensor is modelled separately because it introduces delays and affects the accuracy of the measured output. Each block contains a transfer function that describes that component's behavior. The controller has a transfer function with the variable K_I. The purpose of control design is to select K_I to ensure the SASO properties. This is done by finding the end-to-end transfer function from the reference

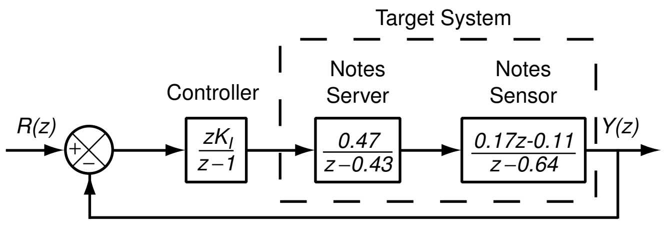
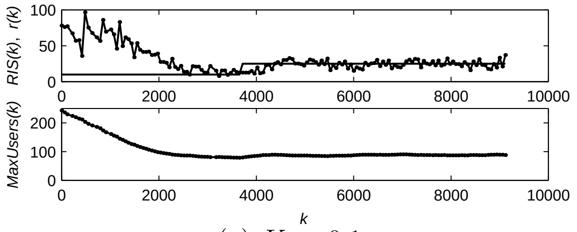


Figure 6: *Block diagram for integral control of the IBM Lotus Domino Server.*

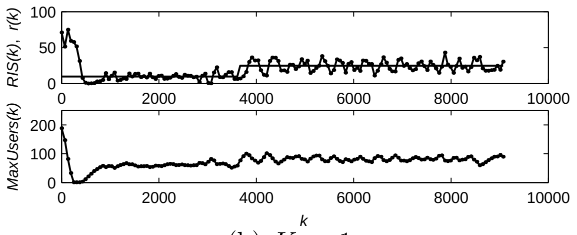
input to the measured output:

$$\frac{zK_I(0.47)(0.17z - 0.11)}{(z-1)(z-0.43)(z-0.64) + zK_I(0.47)(0.17z - 0.11)} \tag{3}$$

By setting K_I to different values, we obtain different poles. For example, if $K_I = 0.1$ in Equation (3), the largest pole is very close to 1, which results in a long settling time. However, if $K_I = 1$, the largest pole is approximately 0.8, which greatly reduces settling times. These effects are clear in Figure 7. More details can be found in [14].



(a) $K_I = 0.1$



(b) $K_I = 1$

Figure 7: *Transient response of the control system in Figure 6. Figure 7(a) has a pole close to 1 and so it converges slowly. Figure 7(b) has a much smaller (but still positive) pole, and so converges quickly.*

As illustrated in this section, control theory provides rigorous analysis and design techniques for building self-managing systems. Both the response to system dynamics (e.g., transient behavior or sensor delay) and robustness to uncertainties (e.g., workload variations or measurement noise) can be studied analytically. Besides the classical control theory used in this section, different formal control methods such as state space optimal control, model reference adaptive control, gain scheduling, and stochastic control can either be deployed directly or used as a source of inspiration in building the analysis and planning components.

3 Building Self-managing Systems Using Control Theory

This section describes how control theory can be used to build self-managing systems and identifies challenges in doing so. The material is structured in terms of the components of the autonomic computing architecture in Figure 1.

3.1 Modeling Resource Dynamics

It should be apparent from the methodology described in Section 2.3, that control analysis and design is based on the ability to model resources. This can be approached in several ways, although the construction of system models for resources remains a significant challenge in the successful application of control theory to computing systems.

Considered first is a purely empirical approach that employs curve fitting to construct static system models; these models do not address dynamics and typically only characterize single input single output relationships. This has been very effective in IBM's mainframe systems [21].

The second approach to modeling is a black box methodology (a.k.a., system identification) such as that described in Section 2.3. This approach requires: choosing an operating point, designing appropriate experiments, and developing empirical models. This approach explicitly models system dynamics and is readily generalized to multiple control inputs and measured outputs. For example, in the Apache HTTP Server, there are two control inputs, the maximum number of clients (denoted by `MaxClients`) that controls the level of concurrency, and the keep alive timeout (denoted by `KeepAlive`) that specifies how long a connection to the server persists after the completing of the last request on that connection. We consider two measured outputs, the utilization of the CPU (denoted by `CPU`) and the utilization of memory (denoted by `MEM`). Since `MaxClients` affects both `CPU` and `MEM` but `KeepAlive` only affects `CPU`, we use the following model

$$\begin{aligned} y_{\mathrm{CPU}}(k) &= a_{\mathrm{CPU}} y_{\mathrm{CPU}}(k-1) + b_{\mathrm{CPU}}\mathrm{KA}(k-1) \\ &+ c_{\mathrm{CPU}}\mathrm{MC}(k-1) \\ y_{\mathrm{MEM}}(k) &= a_{\mathrm{MEM}} y_{\mathrm{MEM}}(k-1) + b_{\mathrm{MEM}}\mathrm{MC}(k-1) \end{aligned}$$

This model works well in studies we have conducted [15].

Still another approach is to develop special purpose models from first principles. An example of this is the first principles analysis done for adaptive queue management in network routers [9]. This approach involves a detailed understanding of the TCP/IP protocol and the development of differential equations to estimate a transfer function.

Note that the above models can be built either off-line or on-line. Although on-line modeling is usually preferred for capturing system and workload variations, special considerations need be given to ensure that the online collected data are sufficient to reflect the system behaviors, a topic called "persistent excitation" that is studied in adaptive control literature.

3.2 Sensors

A significant focus in the management of computing systems is the choice of sensors, especially standardizing interfaces to sensors. The most widely used protocol for accessing sensor data in computing systems is the Simple Network Management Protocol (SNMP) [22]. While this allows for programmatic access, it has not addressed various issues that are of particular concern for control purposes. Among these are the following:

1. Typically, there are multiple measurement sources (even on a single server) that pro-

duce both interval and event data. Unfortunately, the intervals are often not synchronized (e.g., 10 second vs. 1 minute vs. 1 hour), and missing data are common. Even worse, data from different servers often come from clocks that are unsynchronized (or worse still, are synchronized via some complex protocol that converges over a longer window).

2. Often, the metric that it is desirable to regulate is not available. For example, end-to-end response times are notoriously difficult and expensive to obtain. Thus, surrogate metrics are often used such as CPU queue length. Hence, it may well be that the surrogate is well regulated but the desired metric is not.

3. There can be substantial overheads associated with metric collection. For example, it can be quite informative to collect information about the resource consumption of individual requests to a web server. However doing so may consume a substantial fraction of the server CPU. This results in another kind of control problem—determining which measurements to collect and at what frequency.

4. Often, the measurement system has built-in delays. For example, response times cannot be reported until the work unit completes. Sometimes, the mean response time is about the same as the control interval, which can lead to instabilities. Unpredictable delays are common as well since measurement collection is typically the lowest priority task and so is delayed when high priority work arrives (which can be a critical time for the controller).

5. An on-going challenge for developers of instrumentation for computing systems is that there is a wide variation in the semantics of supposedly standard metrics. For example, the metric "paging rate" could mean any of the following: (a) the rate at which pages are written to disk; (b) the rate at which pages are read from disk; and (c) the rate at

which page-ins are requested (not all of which result in accessing secondary storage). Because of these disparities, an attempt has been made to standardize the definition of metrics [23]. However, this effort is limited to UNIX Operating Systems since they have a similar structure and hence similar metrics.

3.3 Effectors

One of the more challenging problems in the control engineering of computing systems is that the set of available effectors (actuators) often has a somewhat complex relationship with the measured output, especially in terms of dynamics. We illustrate this problem by giving several examples.

Consider again the IBM Lotus Domino Server with the objective of controlling the number of active users RIS by adjusting the number of connected users `MaxUsers`. As noted previously, the number of *connected* users is not the same as the number of *active* users. For example, during lunch time, there may be many connected users, very few of whom submit requests. Under these circumstances, `MaxUsers` could be much larger than the number of active users. On the other hand, during busy periods (e.g., close to an end-of-month deadline), almost all connected users may have submitted requests. In this case, `MaxUsers` may be very close to the number of active users.

There is still another complication with `MaxUsers`. The mechanism employed does not maintain a queue of waiting requests to connect to the server. That is, if `MaxUsers` is increased, there is no effect until the next request arrives. If requests are of short duration and are made quickly, there is little delay. However, if requests occur at a lower rate, then this effector introduces a dead time that makes control more challenging.

Another example of a complex effector is the *nice* command used in UNIX systems. *nice* provides a way to adjust the priority of a process, something that is especially important if there is a mixture of CPU intensive and non-CPU intensive work in the system. In theory, *nice* can be used to enforce SLOs dealing with the fraction of the CPU that a process receives. However, this turns out to be complicated to do in practice because of the way *nice* affects priorities. As shown in [24], this is non-linear relationship that depends on the number of processes competing for the CPU as well as the range of priority numbers used. Recognizing the limitations of using *nice*, special purpose schedulers have been developed (e.g., [25]). In essence, these approaches create a new, more rational set of effectors.

A final example is the start-time fair queueing (SFQ) algorithm. This resource management algorithm controls the service delivered by a resource by controlling the priority assigned to incoming requests [26]. Specifically, SFQ operates by tagging incoming work by class, and the tags determine the priority by which the request is processed. Unfortunately, this mechanism has some subtle, load-dependent characteristics that create challenges for designing control systems. In particular, changing the tag assigned to a new request has no effect until the requests ahead of it have been processed. If load is light, there will be few such requests, and so little dead time. However, if loads are heavy, dead times could be substantial. If dead times can be predicted, then compensation might be possible. Otherwise, the control performance of this effector can be impaired, possibility even resulting in stability problems.

4 Case Study

This section gives a case study of applying control theory to computing systems. The results of this work have been incorporated into an IBM database management product. More details can be found in [12].

The operation of many important software systems involves the execution of administrative utilities needed to preserve the system's integrity and efficiency. Such administrative utilities have the following characteristics: (1) their execution is essential to the integrity of the system; (2) however, they can severely impair the performance of the primary function (hereafter, referred to as **production work**) if executed concurrently with that work. Hence, administrators typically use overnight periods, holidays or scheduled downtimes to execute such tasks. With the advent of 24×7 operation, such administrative windows are disappearing, creating a significant problem for the system administrator. Therefore, it is highly desirable to provide enforceable policies for regulating the execution of utilities.

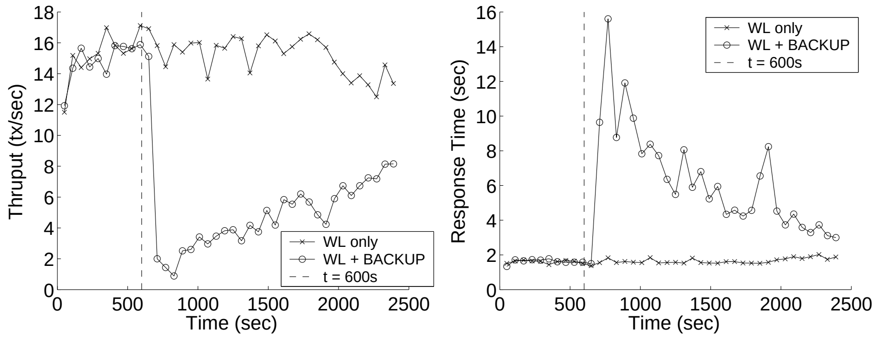


Figure 8: Performance degradation due to running utilities. Plots show time-series data of throughput and response time measured at the client, averaged over a 60s interval.

Fig. 8 demonstrates the dramatic performance degradation from running a database backup utility while emulated clients are running a transaction-oriented workload against that database. The throughput of the system without this backup utility (i. e., workload only) averages 15 transactions per second (tps). When the backup utility is started at t=600sec, the throughput drops to between 25–50% of the original level, and a corresponding increase is seen in the response time. Note also that with the utility running, throughput increases with time (indicating that the resource demands of the utility decrease). Thus, enforcing policies for administrative utilities faces the challenge of dealing with such dynamics.

What kinds of policies should be used to regulate administrative utilities? Based on our understanding of the requirements of database administrators, we believe that policies should be expressed in terms of degradation of production work. A general form for such a policy is

Administrative Utility Performance Policy: There should be no more than an x% performance degradation of production work as a result of executing administrative utilities.

In these policies, the administrator thinks in terms of "degradation units" that are normalized in a way that is fairly independent of the specific performance metric (e.g., response time, transaction rate). It is implicit that the utilities should complete as early as possible within this constraint, i. e., the system should not be unnecessarily idle.

There are two challenges with enforcing such policies.

Challenge 1: Provide a mechanism for controlling the performance degradation from utilities.

We use the term **throttling** to refer to limiting the execution of utilities in some way so as to reduce their performance impact. One example of a possible throttling mechanism is priority, such as `nice` values in Unix systems (although this turns out to be a poor choice, as discussed later).

Challenge 2: Translate from degradation units (specified in the policy) to throttling units (understood by the mechanism).

Such translation is essential so that administrators can work in terms of their policies, not the details of the managed system. Unfortunately, accomplishing this translation is complicated by the need to distinguish between performance degradation of the production work caused by contention with the administrative utilities and changes in the production work itself (e.g., due to time-of-day variations).

We begin by addressing the first challenge. One approach is to use operating system (OS) priorities, an existing capability provided by all modern operating systems. Throttling could be achieved by making the utility threads less preferred than threads doing production work. In principle, such a scheme is appealing in that it does not require modifications to the utilities. However, it does require that the utility executes in a separate dispatchable unit (process/thread) to which the OS assigns priorities. Also, a priority-based scheme requires that access to *all* resources be based on the same priorities. Unfortunately, the priority mechanisms used in most variants of Unix and Windows only affect CPU scheduling. Such an approach has little impact on administrative utilities that are I/O bound (e.g., backup).

Our approach is to use self-imposed sleep (SIS). SIS relies on another OS service: a sleep system call which is parameterize by a time interval. Most modern OSes provide some version of a sleep system call that makes the process or thread not schedulable for the specified interval. Fig. 9 describes a throttling API that uses this sleep service.

```
FUNCTION Utility()
BEGIN
    WHILE (NOT done)
    BEGIN
        ...  do some work ...
        SleepIfNeeded()
    END
END
```

(a) Inserting SIS point

```
FUNCTION SleepIfNeeded()
BEGIN
    (workTime, sleepTime) = GetThrottlingLevel() ;
    timeWorked = Now() - workStart ;
    IF (timeWorked > workTime)
        SLEEP( sleepTime ) ;
        workStart = Now() ;
    ENDIF
END
```

(b) SIS implementation

Figure 9: High-level utility structure and sleep point insertion

We address the second challenge by using a feedback control system to translate degradation units (specified in the policy) into throttling units. This should be done in a manner that not only achieves the administrative target of "x% performance degradation" but also adapts quickly to changes in the resource requirements of utilities and/or production work.

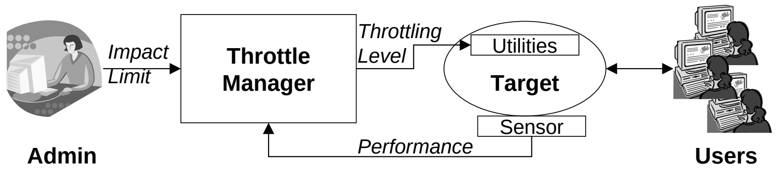


Figure 10: Throttling system operation

The overall operation of our proposed automated throttling system is illustrated in Fig. 10. Administrators specify the degradation limit, which corresponds to the x in the policy described above. The main component is the Throttle Manager, which determines the throttling levels (i. e., sleep fraction) for the utilities based on the degradation limit as well as performance metrics from the target system.

The sensor in Fig. 10 estimates the performance degradation due to utilities. This is done by first estimating the metric's **baseline**, which is the value of the metric if there were no utilities running. The baseline value is compared to the most recent performance feedback to calculate the current degradation (as a fraction):

$$Degradation = 1 - \frac{performance}{baseline}$$

Given the current degradation level, the Throttle Manager must calculate throttling levels for the utilities. Because of the relatively straightforward effects of `sleepTime` on performance, we use a standard Proportional-Integral (PI) controller from linear control theory[27] to drive this error quantity to zero, thereby enforcing the throttling policy. A PI control structure is proven to be very stable and robust and is guaranteed to eliminate any error in steady-state. It is used in nearly 90% of all controller applications in the real world. A new throttling value at time $k + 1$ is computed as follows:

$$throttling(k + 1) = K_P * error(k) + K_I * \sum_{i=0}^{k} error(i) \tag{4}$$

K_P and K_I are chosen to ensure the SASO properties.

To evaluate our control system, we first show in Fig. 11(a) that the throttling system follows the policy limit in the case of a steady workload generated by 25 emulated users. For comparison, the workload performance as well as the effect of an unthrottled utility (from Fig. 8) are also shown. While the average throughput without the BACKUP running is 15.1 tps, the throughput with a throttled BACKUP is 9.4 tps – a degradation of 38%, which is close to the desired 30%. Note how the throttling system compensates for the decreasing resource demands of the utility by lowering the sleep fraction (Fig. 11(c)), resulting in a throughput profile that is more parallel to the no-utility case.

To highlight the adaptive nature of this system, we consider a scenario where there is a surge in the number of users accessing the database system while the BACKUP utility is executing. We start with a nominal workload consisting of 10 emulated users, and start the utility at 300 sec. At time 1500 sec, an additional 15 users are added (thus, resulting in a total of 25 users). Fig. 11(b) shows the raw performance data for the surge, with the no-utility case (in the same scenario) shown for reference. We see that the throttling system adapts when the workload increases, reaching a new throttling level within 600sec. For this case, the pre-surge average throughputs are 13.1 (workload only) and 8.37 (throttled BACKUP) – a degradation of 36%. Analogously, the post-surge degradation is 19%. Note that the sleep fraction used (and the resultant throughput) towards the latter half of the run is similar to the value seen for the steady-workload case, indicating that the models learned by the Baseline Estimator are similar.

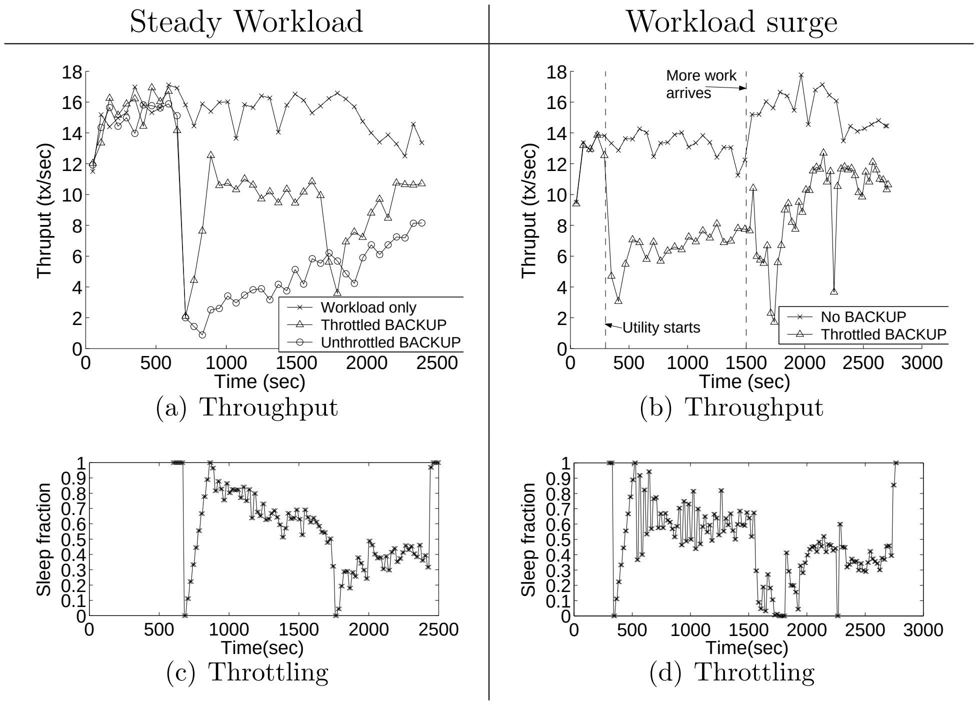


Figure 11: Effect of throttling a utility under a steady workload and a workload surge with a 30% impact policy. x axis is time. The throughput data shown is computed over 1 minute intervals, sleep fraction is set every 20sec.

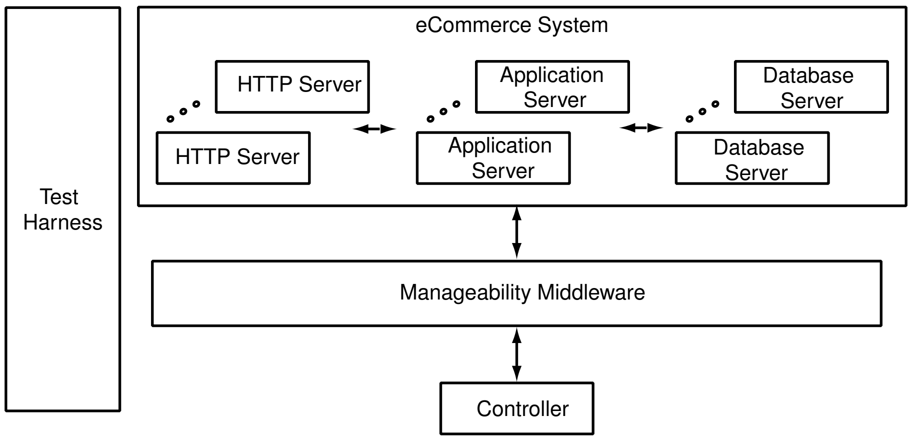


Figure 12: *Architecture of the Deployable Testbed for Autonomic Computing (DTAC).*

5 Deployable Testbed for Autonomic Computing (DTAC)

Section 3 details a number of challenges with building autonomic systems. Some of these challenges relate to developing appropriate control techniques. Other challenges relate to engineering the software components that support the requirements of control systems. Unfortunately, to evaluate efforts in either area, a complete system must be developed, a fact that hinders progress since researchers prefer to focus on their area of expertise. For example, it took months to develop the system in Section 4.

The foregoing has motivated our interest in DTAC, a deployable testbed for autonomic computing. DTAC is intended to be a complete end-to-end system with pluggable components so as to facilitate research in various aspects of autonomic computing. For example, researchers focusing on control algorithms need only modify these components, but they would still have an end-to-end system to evaluate their algorithms. Similarly, researchers primarily interested in sensors and effectors could replace these elements and take advantage of existing control algorithms.

Figure 12 displays our initial architecture for DTAC. There are four layers in the architecture, all of which are intended to be pluggable. The Test Harness provides the overall experimental environment, including the generation of synthetic workload, and a suite of tools for analyzing experimental results. The operation of the Testbed is as follows: (1) The Test Harness creates a request that is sent to an HTTP server; (2) HTTP servers process requests, forwarding to an Application Server those requests that require extensive processing; and (3) Application Servers forward to a Database Server those requests that require data intensive operations. To satisfy scaling requirements, one or more tier of the eCommerce System may contain server clusters with appropriate load balancing.

In terms of the autonomic computing architecture, the eCommerce system is a set of resources. These resources have a variety of sensors for accessing measurements and effectors for controlling their behavior. For example, effectors of interest in the Apache HTTP server include the `KeepAlive` timeout and the maximum number of clients. Key effectors for the Database server might be the size of memory pools for sorts and joins. Observe that there is a natural hierarchy of resources that may well imply a hierarchy of managers of these resources. That is, the full eCommerce system provides statistics on end-to-end response times and its main effector is based on traffic shaping. In addition, there may be managers for clusters of servers that provide sensors for relative utilizations of servers within the cluster and an effector that determines how to balance load among servers within the cluster.

The variety of different sensors and effectors motivates the need for Manageability Middleware that virtualizes these differences and provides commonly used functions. In terms of the autonomic computing architecture, this corresponds to the monitoring and execution components in the autonomic manager. Examples of Manageability Middleware include KinestheticsExtreme [4], IBM's autonomic computing Toolkit [28], and ControlWare [29]. We expect that the Manageability Middleware will incorporate common functions, such as filtering events and maintaining state.

The Controller Layer is primarily responsible for making decisions and taking actions (although this layer may incorporate elements of analysis as well). This is the primary layer for doing policy interpretation and enforcement.

Last, the Test Harness operates the experimental environment, including the control of experimental runs, workload generation, data collection, and reporting. Workload generation is of particular concern since it has a dramatic effect on the experimental results. We advocate the use of industry standard workloads, such as those developed by the Transaction Processing Council (TPC) and the Standard Performance Evaluation Corporation (SPEC).

We emphasize that Figure 12 depicts the layers in our Testbed, not necessarily component instances. For example, there may be separate instances of Manageability Middleware for each server, along with their own Controller. And there may be separate instances of Manageability Middleware and Controllers for each server cluster.

Our goal is to develop an easily deployable package that instantiates the above architecture in a way that researchers can readily substitute their components and run experiments to evaluate their technologies. For the eCommerce System, we plan to use the Apache HTTP Server, the Tomcat Application Server, and the MySQL Database Server. All are publically available, both the executables and the source. Also, they are widely used in production systems.

Going a step further, we anticipate that a *DTAC stack* will be needed on each resource to be managed in the eCommerce System. Figure 13 depicts at a high level the layers in this stack, beginning with the lowest level – *level 0*, which are the resources (applications) to be managed.

Layer 1 are the *touchpoints*. There is a set of touchpoints associated with every resource to be managed. Touchpoints make up *level 1* and perform the duties of sensors and effectors for the resource of interest, they encapsulate resource specific strategies for interacting with the resource.

Layer 2 contains the resource models that specify "interesting" facets and attributes of a resource. These facets include considerations for performance, configuration, and operating environment.

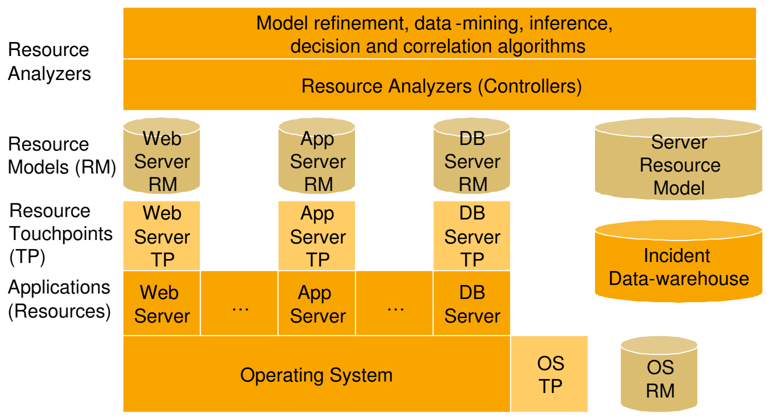


Figure 13: *Software stack needed by managed elements to run DTAC.*

Level 3 consists of the resource analyzers. Resource analyzers provide filtering, control analysis, root cause analysis, forecasting, proactive management, and other higher level functions by using technologies such as inference engines, data-mining and correlation algorithms.

The layers in the DTAC stack have associated interfaces as well. These are:

- ITouchpoint – the interface implemented by elements performing the duties of sensors and effectors
- IResourceModel – the interface implemented by elements that aggregate data of interest about a resource
- IAnalyzer – the interface implemented by elements that refines the resource model and makes decisions over it

We are in the early stages of discussion of the choice of Manageability Middleware and Controller to distribute with the Testbed. The intent is to use something simple. For example, the Controller distributed with the Testbed might be a classical multiple input multiple output (MIMO) controller (e.g., [15]) that manipulates configuration parameters in all three tiers. More generally, there are two main requirements for components in the Testbed package. First, the component should be sufficient to conduct experiments on unrelated components. Second, components distributed with the Testbed should illustrate the use of the APIs required for component pluggability.

The details of the Test Harness are still under consideration. However, the workload driver will likely be the TPC Web Workload (TPC-W) [30] since there is a publically available software driver.

6 Conclusions

This paper takes the position that control theory can provide an architectural and analytic foundation for building self-managing systems. Our approach to establishing the architectural connection is to show the correspondence between the elements of the IBM autonomic computing architecture and those in the elements control systems. For example, the sensors and effectors of the autonomic architecture provide the measured outputs and control inputs in control systems. The benefit of making this connection is that control theory provides a rich set of methodologies for building automated with properties such as stability, short settling times, and accurate regulation. This said, there remain considerable challenges in applying control theory to computing systems, such as developing reliable resource models, handling sensor delays, and addressing lead times in effector actions. These challenges motivate the need for broader engagement of the research community in these areas.

The last observation has motivated our recent efforts with developing DTAC, a deployable testbed for autonomic computing. DTAC is intended to support the study of a wide range of research problems related to automating the management of distributed systems. Examples of research questions include: (1) What sensors and effectors work best in maintaining service level objectives? This can be investigated by modifying one or more of the eCommerce tiers. (2) How can data from heterogeneous sensors be virtualized in a way that supports the goal of end-to-end service level management? This can be studied by developing appropriate

Manageability Middleware. (3) Which control techniques best ensure end-to-end service level objectives? This may involve a combination of pluggable Controllers and selection of different sensors and effectors. (4) What is required to better automate provisioning of distributed systems? This may entail modifications to the eCommerce, Manageability Middleware, and Controller layers.

7 Acknowledgements

Kaiser's Programming Systems Lab is funded in part by National Science Foundation grants CNS-0426623, CCR-0203876 and EIA-0202063.

References

[1] A. Fox and D. Patterson, "Self-repairing computers," in *Scientific American*, May 2003.

[2] K. Milliken, A. Cruise, R. Ennis, A. Finkel, J. Hellerstein, D. Loeb, D. Klein, M. Masullo, H. V. Woerkom, and N. Waite, "Yes/mvs and the autonomation of operations for large computer complexes," *IBM Systems Journal*, vol. 25, no. 2, 1986.

[3] I. B. M. Corporation, "An architectural blueprint for autonomic computing," Tech. Rep. http://www-306.ibm.com/autonomic/pdfs/ACwpFinal.pdf, IBM Corporation, 2004.

[4] G. Kaiser, J. Parekh, P. Gross, and G. Valetto, "Kinesthetics extreme: An external infrastructure for monitoring distributed legacy systems," in *Fifth Annual International Active Middleware Workshop*, 2003.

[5] S. Keshav, "A control-theoretic approach to flow control," in *Proceedings of ACM SIGCOMM '91*, Sept. 1991.

[6] K. Li, M. H. Shor, J. Walpole, C. Pu, and D. C. Steere, "Modeling the effect of short-term rate variations on tcp-friendly congestion control behavior," in *Proceedings of the American Control Conference*, pp. 3006–3012, 2001.

[7] E. Altman, T. Basar, and R. Srikant, "Congestion control as a stochastic control problem with action delays," *Automatica*, vol. 35, pp. 1936–1950, 1999.

[8] C. V. Hollot, V. Misra, D. Towsley, and W. B. Gong, "On designing improved controllers for AQM routers supporting TCP flows," in *Proceedings of IEEE INFOCOM '01*, (Anchorage, Alaska), Apr. 2001.

[9] C. V. Hollot, V. Misra, D. Towsley, and W. B. Gong, "A control theoretic analysis of RED," in *Proceedings of IEEE INFOCOM '01*, (Anchorage, Alaska), Apr. 2001.

[10] T. F. Abdelzaher and N. Bhatti, "Adaptive content delivery for Web server QoS," in *International Workshop on Quality of Service*, (London, UK), June 1999.

[11] Y. Lu, A. Saxena, and T. F. Abdelzaher, "Differentiated caching services: A control-theoretic approach," in *International Conference on Distributed Computing Systems*, Apr. 2001.

[12] S. Parekh, K. Rose, J. L. Hellerstein, S. Lightstone, M. Huras, and V. Chang, "Managing the performance impact of administrative utilities," in *IFIP Conference on Distributed Systems Operations and Management*, 2003.

[13] L. Sha, X. Liu, Y. Lu, and T. Abdelzaher, "Queueing model based network server performance control," in *IEEE RealTime Systems Symposium*, 2002.

[14] S. Parekh, N. Gandhi, J. Hellerstein, D. Tilbury, J. Bigus, and T. S. Jayram, "Using control theory to acheive service level objectives in performance management," *Real-time Systems Journal*, vol. 23, pp. 127–141, 2002.

[15] Y. Diao, N. Gandhi, J. L. Hellerstein, S. Parekh, and D. Tilbury, "Using MIMO feedback control to enforce policies for interrelated metrics with application to the Apache Web server," in *IEEE/IFIP Network Operations and Management*, April 2002.

[16] Y. Diao, J. L. Hellerstein, and S. Parekh, "Optimizing quality of service using fuzzy control," in *Distributed Systems Operations and Management*, 2002.

[17] Y. Diao, J. L. Hellerstein, A. Storm, M. Surendra, S. Lightstone, S. Parekh, and C. Garcia-Arellano, "Using MIMO Linear Control for Load Balancing in Computing Systems," in *American Control Conference*, June 2004.

[18] Y. Diao, J. L. Hellerstein, G. Kaiser, S. Parekh, and D. Phung, "Self-managing systems: A control theory foundation," in *Engineering of Autonomic Systems*, April 2005.

[19] C. Lu, J. A. Stankovic, T. F. Abdelzaher, G. Tao, S. H. Son, and M. Markley, "Performance specifications and metrics for adaptive real-time systems," in *Proceedings of the IEEE Real Time Systems Symposium*, (Orlando), 2000.

[20] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury, *Feedback Control of Computing Systems.* John Wiley & Sons, 2004.

[21] J. Aman, C. K. Eilert, D. Emmes, P. Yocom, and D. Dillenberger, "Adaptive algorithms for managing a distributed data processing workload," *IBM Systems Journal*, vol. 36, no. 2, 1997.

[22] A. Tannenbam, *Computer Networks.* Prentice Hall, 4th ed., 2002.

[23] T. O. Group, *Systems Management: The Universal Measurement Architecture.* The Open Group, 1997.

[24] J. L. Hellerstein, "Achieving service rate objectives with decay usage scheduling," *IEEE Transactions on Software Engineering*, vol. 19, no. 8, pp. 813–825, 1993.

[25] J. Kay and P. Lauder, “A fair share scheduler,” *Communications of the ACM*, vol. 31, no. 1, pp. 44–55, 1988.

[26] P. Goyal, H. M. Vin, and H. Cheng, “Start-time fair queueing: A scheduling algorithm for integrated servicepacket switching networks,” in *ACM SIGCOMM*, 1996.

[27] K. Ogata, *Modern Control Engineering*. Prentice Hall, 3rd ed., 1997.

[28] IBM, “Autonomic computing toolkit,” Tech. Rep. http://www-106.ibm.com/developerworks/autonomic/probdet.html, IBM, 2004.

[29] R. Zhang, C. Lu, T. F. Abdelzaher, and J. A. Stankovic, “Controlware: A middleware architecture for feedback control of software performance,” in *Internation Conference on Distributed Computing Systems*, pp. 301–310, 2002.

[30] TPC, “TPC Web,” Tech. Rep. http://www.tpc.org/tpcw, Transaction Processing Council, 2004.