Except for formatting, this report is identical to the paper published in IEEE Journal on Selected Areas in Communications, Volume 23, Number 12, December 2005. It was retroactively submitted as a Columbia Computer Science technical report on May 10, 2007 and is assigned a technical report number and date reflecting its original publication date.

A Control Theory Foundation for Self-Managing Computing Systems

Yixin Diao, *Member, IEEE*, Joseph L. Hellerstein, *Senior Member, IEEE*, Sujay Parekh, *Student Member, IEEE*, Rean Griffith, Gail E. Kaiser, *Senior Member, IEEE*, and Dan Phung

***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. Control theory provides a rich set of methodologies for building automated self-diagnosis and self-repairing systems with properties such as stability, short settling times, and accurate regulation. However, there are challenges in applying control theory to computing systems, such as developing effective resource models, handling sensor delays, and addressing lead times in effector actions. We propose a deployable testbed for autonomic computing (DTAC) that we believe will reduce the barriers to addressing research problems in applying control theory to computing systems. The initial DTAC architecture is described along with several problems that it can be used to investigate.**

***Index Terms*—Actuator, closed loop control, dynamics, resource management, sensor, testbed.**

I. 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.

Manuscript received June 30, 2005; revised July 20, 2005. The work of the Programming Systems Laboratory is supported in part by the National Science Foundation under Grant CNS-0426623, Grant CCR-0203876, and Grant EIA-0202063, and in part by Microsoft Research.

Y. Diao, J. L. Hellerstein, and S. Parekh are with the IBM Thomas J. Watson Research Center, Hawthorne, NY 10532 USA (e-mail: diao@us.ibm.com; hellers@us.ibm.com; sujay@us.ibm.com).

R. Griffith, G. E. Kaiser, and D. Phung are with the Computer Science Department, Columbia University, New York, NY 10027-7003 USA (e-mail: rg2023@cs.columbia.edu; kaiser@cs.columbia.edu; phung@cs.columbia.edu).

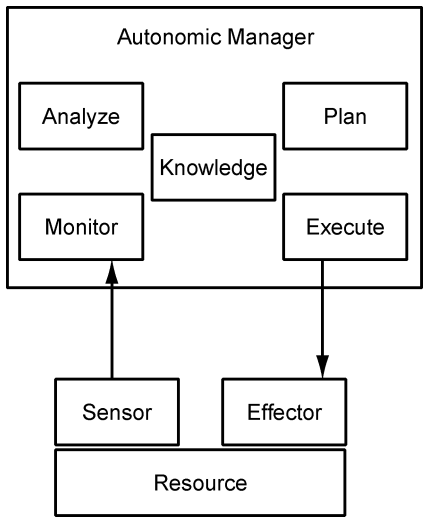


Fig. 1. Architecture for autonomic computing.

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]). Fig. 1 depicts the components and key interactions for a single autonomic manager and a single resource. The resource (sometimes 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 transmission control protocol (TCP) (e.g., [6]) and some have considered 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] and [9]). Control theory has also 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 limiting the impact of administrative utilities on production work [12]. Examples of regulating resource utilizations include a mixture of queueing and control theory used to regulate Apache HTTP Server [13], regulation of the IBM Lotus Domino Server [14], and multiple-input–multiple-output (MIMO) control of 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]. Here, we go beyond our earlier work by including a case study of applying control theory to an IBM software product, and we include more details on a deployable testbed for research in the application of control theory to autonomic computing. Specifically, Section II seeks to educate systems oriented computer science researchers and practitioners on the concepts and techniques needed to apply control theory to computing systems. Section III presents a case study of applying control theory to an IBM database management product. Section IV proposes a deployable testbed for autonomic computing (DTAC) that is intended to foster research that addresses the challenges identified. Our conclusions are contained in Section V.

II. Control Theoretic Framework

This section relates control theory to self-managing systems.

A. Components of a Control System

Over the last 60 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 Fig. 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 Fig. 2. These elements are the following.

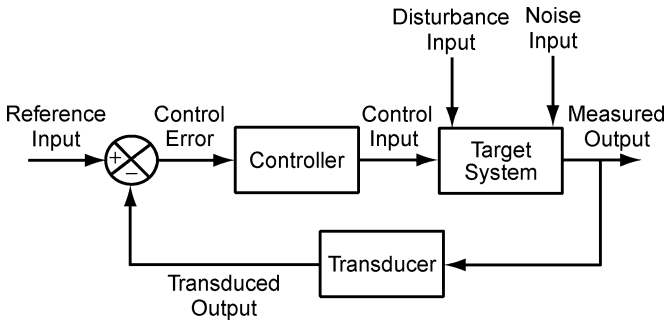


Fig. 2. Block diagram of a feedback control system. The reference input is the desired value of the system's measured output. The controller adjusts the setting of control input to the target system so that its measured output is equal to the reference input. The transducer represents effects such as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

- Target system, which is the computing system to be controlled.
- 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 Apache HTTP Server).
- Measured output, which is a measurable characteristic of the target system such as CPU utilization and response time.
- Disturbance input, which is any change that affects the way in which the control input influences the measured output of the target system (e.g., running a virus scan or a backup).
- Noise input, which is any effect 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.
- Transducer, which transforms the measured output so that it can be compared with the reference input (e.g., smoothing stochastics of the output).
- Control error, which is the difference between the reference input and the measured output (which may include noise and/or may pass through a transducer).
- 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.

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 on 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.

B. 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 the following.

- Regulation: 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 regulation 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 Apache HTTP Server so as to minimize response times.

We have found several properties of feedback control systems to be of interest in computing systems. A control system is *stable* if for any bounded input, the output is also bounded. Since no real world system produces an unbounded output, in practice, unstable systems produce large oscillations, especially limit cycles that alternate between extreme values of metrics. A control system is *accurate* if the measured output converges (or becomes sufficiently close) to the reference input, such as ensuring that throughput is maximized without exceeding response time constraints. A control system has *short settling times* if it converges quickly to its steady-state value, which is often important if there are time-varying workloads. Finally, *overshoot* is an important consideration in control systems if there are threshold effects such as buffer overflows.

One appeal of control theory is that it provides a framework to analyze and design closed loop systems 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, unstable systems have large response times and/or low throughputs, characteristics that can make the system unusable.

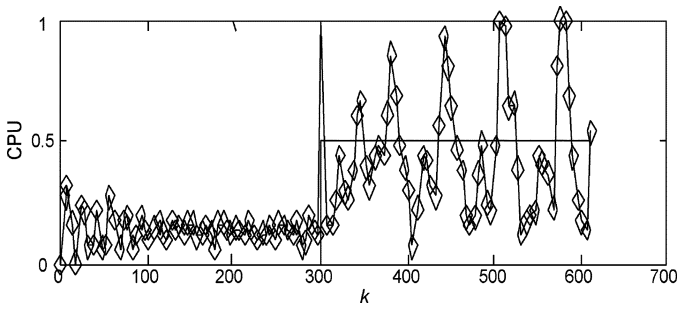


Fig. 3. Example of an unstable feedback control system for the Apache HTTP Server. The instability results from having an improperly designed controller.

Fig. 3 displays an instability in an 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 s, the system operates without feedback control. When the controller is turned on, a reference input of 0.5 is used. 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 Fig. 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 (typically, within 2%). 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 [19].

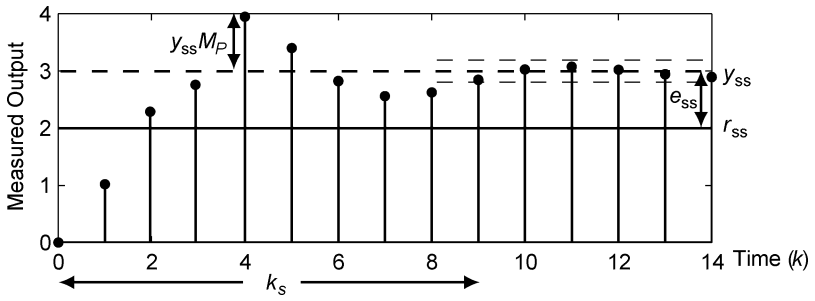


Fig. 4. Response of a stable system to a step change in the reference input. At time 0, the reference input changes from 0 to 2. The system reaches steady-state when its output always lie between the light weight dashed lines. Depicted are the steady-state error (e_{ss}), settling time (k_s), and maximum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

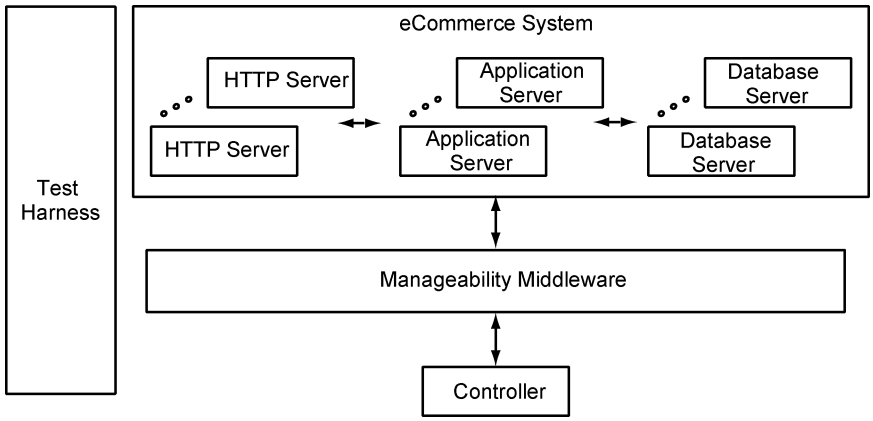


Fig. 5. Architecture of the DTAC.

C. 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 kth value of the former and $y(k)$ to denote the kth value of the latter. (Actually, $u(k)$ and $y(k)$ are offsets from a desired operating point.) An e-mail access and update 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 1 min interval. We construct the following simple autoregressive model using least squares regression:

$$y(k+1) = 0.43y(k) + 0.47u(k). \quad (1)$$

Fig. 6(a) displays the values of $u(k)$ (solid line) and the corresponding $y(k)$ ("x"s). From Fig. 6(b), we see that the model fits these data quite well in that there is a close correspondence between observed RIS and predicted RIS.

To better facilitate control analysis, (1) is put into the form of a transfer function. A transfer function describes how inputs such as `MaxUsers` are transformed into outputs such as RIS. A transfer function is represented as a Z transform, a mathematical construct that provides a compact specification of time-varying functions in terms of the time shift operator z. The transfer function of (1) is

$$\frac{0.47}{z - 0.43}. \quad (2)$$

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, (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 (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 (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

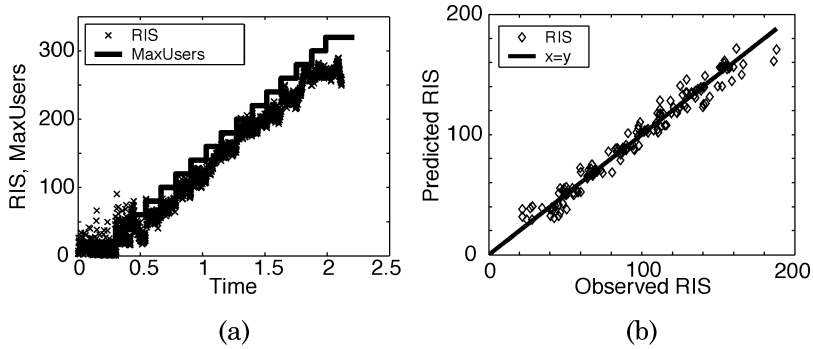


Fig. 6. Data and model evaluation for the IBM Lotus Domino Server. (a) Data used in system identification. (b) Model evaluation.

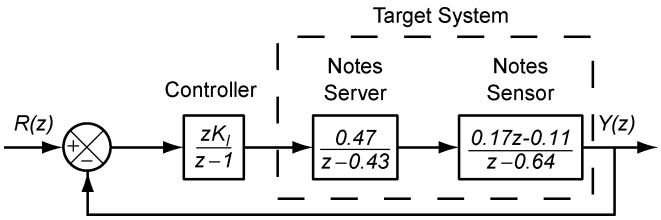


Fig. 7. Block diagram for integral control of the IBM Lotus Domino Server.

approximately $-4/ln|p|$ [19]. Thus, the settling time for (2) is approximately 5 time units.

Fig. 7 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 modeled 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 transfer function from the reference 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 (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 Fig. 8. More details can be found in [14].

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.

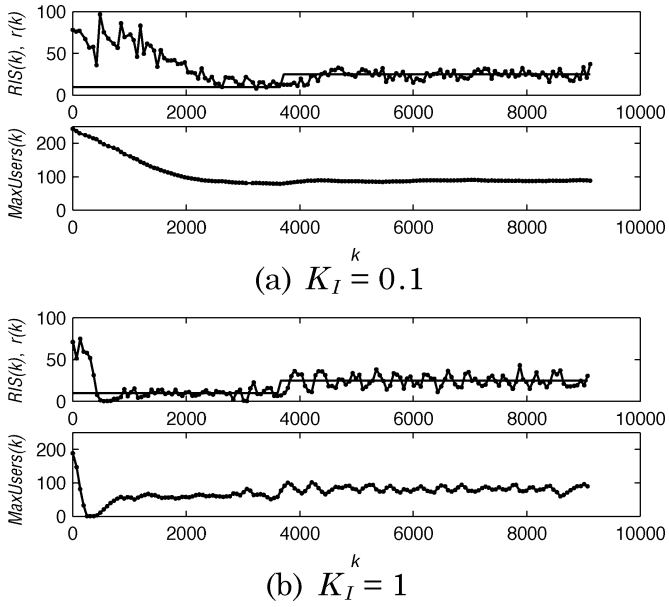


Fig. 8. Transient response of the control system in Fig. 7. (a) Has a pole close to 1 and so it converges slowly. (b) Has a much smaller (but still positive) pole, and so converges quickly.

III. 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].

Fig. 9 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 $t = 600$ s, 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

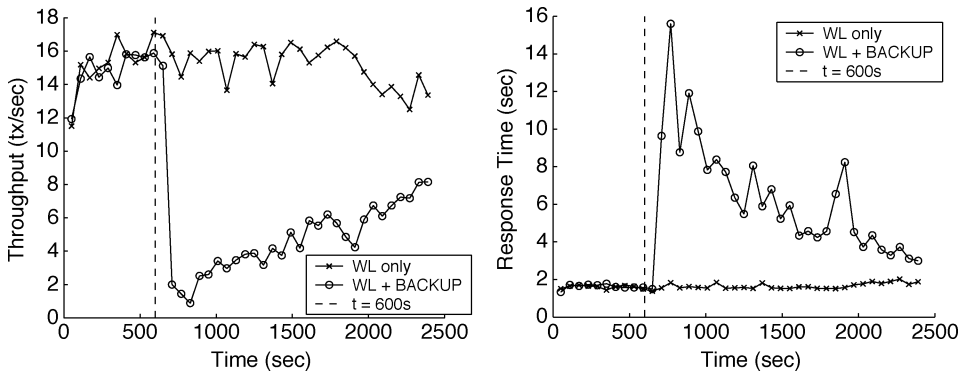


Fig. 9. Performance degradation due to running utilities. Plots show time-series data of throughput and response time measured at the client, averaged over a 60 s interval.

database administrators, we believe that policies should be expressed in terms of degradation of production work. A specific instance of 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 and 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 OSs. 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).

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

(a)

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

(b)

Fig. 10. High-level utility structure and sleep point insertion. (a) Inserting SIS point. (b) SIS implementation.

Our approach is to use self-imposed sleep (SIS). SIS relies on another OS service: a sleep system call which is parameterized 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. 10 describes a throttling API that uses this sleep service.

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.

The overall operation of our proposed automated throttling system is illustrated in Fig. 11. Administrators specify the degradation limit, which corresponds to the x in the policy described above. The main component is the Throttle Manager,

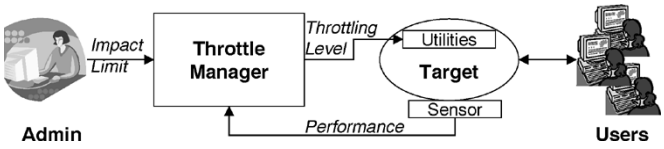


Fig. 11. Throttling system operation.

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. 11 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 with the most recent performance feedback to calculate the current degradation (as a fraction)

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

Note that the baseline estimator changes over time due to the dynamics of workloads and other factors.

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 [20] 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:

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

K_P and K_I are chosen to ensure the SASO properties.

To evaluate our control system, we first show in Fig. 12(a) that the throttling system follows the policy limit in the case of a steady workload generated by 25 emulated users with the introduction of a BACKUP job at time 600. For comparison, the workload performance as well as the effect of an unthrottled utility (from Fig. 9) 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. 12(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 ten emulated users, and start the utility at 300 s. At time 1500 s, an additional 15 users are added (thus, resulting in a total of 25 users). Fig. 12(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 600 s. For this case, the presurge average throughputs are 13.1 (workload only) and 8.37 (throttled BACKUP)—a degradation of 36%. Analogously, the postsurge degradation is 19%. Note that the sleep fraction used (and the resultant throughput) toward 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.

IV. Deployable Testbed for Autonomic Computing (DTAC)

Some of the challenges of building self-managing systems relate to developing appropriate control techniques. Other challenges relate to engineering the software components that support the requirements of control systems, such as the development of appropriate sensors and effectors. 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 III.

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.

The target system in DTAC is a multitiered e-commerce system because of its widespread use in industry and the availability of open source software. Fig. 5 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 e-commerce system may contain server clusters with appropriate load balancing.

In terms of the autonomic computing architecture [3], the e-commerce 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. Further, there may be aggregations of sensors and effectors due to the hierarchical of resources since such a structure may well imply a hierarchy of managers of these resources. For example, the e-commerce system may provide statistics on end-to-end response times and its main effector may be based

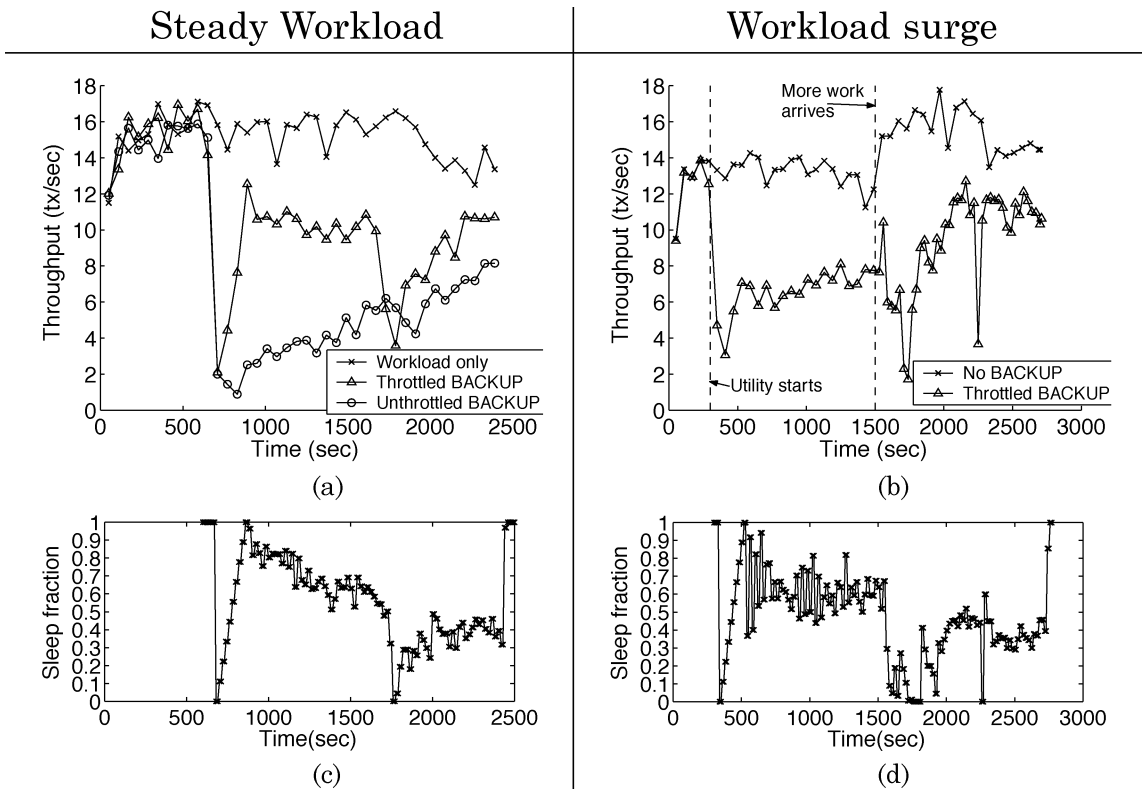


Fig. 12. Effect of throttling a utility under a steady workload and a workload surge with a 30% impact policy. x axis is time. The throughput data shown is computed over 1 min intervals, sleep fraction is set every 20 s. (a) and (b) Throughput. (c) and (d) Throttling.

on traffic shaping, a control that affects arrival rates at all tiers in the e-commerce system.

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 Kinesthetics Extreme [4], IBM's autonomic computing toolkit [21], and Controlware [22]. We expect that the manageability middleware will incorporate common functions, such as filtering events and maintaining state. However, virtualizing this layer of management requires a standard way to describe resources, events, and other artifacts produced by target systems.

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.

Finally, 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 Fig. 5 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 e-commerce 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 e-commerce system. Fig. 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.

Layer 3 consists of the resource analyzers. Resource analyzers provide filtering, control analysis, root cause analysis, forecasting, proactive management, and other higher level

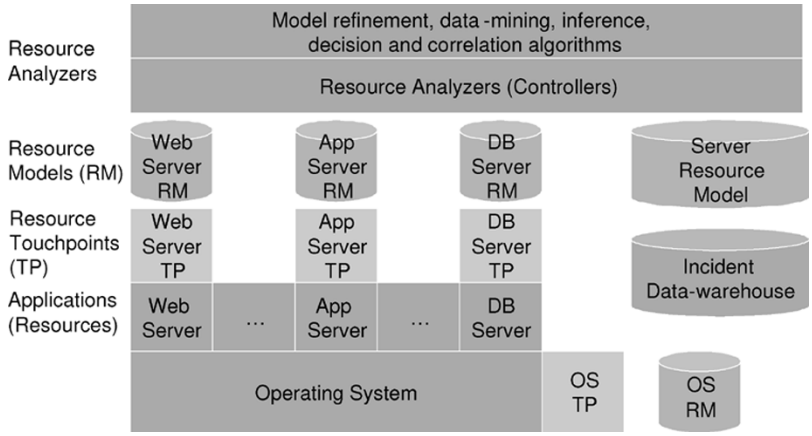


Fig. 13. Software stack needed by managed elements to run DTAC.

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 the following:

- 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 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) [23] since there is a publically available software driver.

V. Conclusion

This paper takes the position that control theory can provide an architectural and analytic foundation for building self-managing systems. Indeed, we show that there is a correspondence between the elements of the IBM autonomic computing architecture and the elements in control systems. There remain considerable challenges in applying control theory to computing systems, such as developing reliable resource models, handling sensor delays, addressing lead times in effector actions, and benchmarking. To engage the research community in addressing these challenges, we are developing a DTAC. DTAC is intended to be a reference environment in which to study practical control issues for computing systems, such as: the choice of sensors and effectors, the choice of control techniques to ensure end-to-end service level objectives, and ways to better automate provisioning in distributed systems.

References

[1] A. Fox and D. Patterson, "Self-repairing computers," *Sci. Amer.*, pp. 54–61, 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 Syst. J.*, vol. 25, no. 2, pp. 159–180, 1986.

[3] IBM-Corporation. An architectural blueprint for autonomic computing. [Online]. Available: http://www-03.ibm.com/autonomic/pdfs/ACwp-Final.pdf

[4] G. Kaiser, J. Parekh, P. Gross, and G. Valetto, "Kinesthetics extreme: An external infrastructure for monitoring distributed legacy systems," in *Proc. 5th Annu. Int. Active Middleware Workshop*, 2003, pp. 22–30.

[5] S. Keshav, "A control-theoretic approach to flow control," in *Proc. ACM SIGCOMM*, Sep. 1991, pp. 3–15.

[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 *Proc. Amer. Control Conf.*, 2001, pp. 3006–3012.

[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 *Proc. IEEE INFOCOM*, Anchorage, AK, Apr. 2001, pp. 1726–1734.

[9] ——, "A control theoretic analysis of RED," in *Proc. IEEE INFOCOM*, Anchorage, AK, Apr. 2001, pp. 1510–1519.

[10] T. F. Abdelzaher and N. Bhatti, "Adaptive content delivery for Web server QoS," *Comput. Netw.*, vol. 31, pp. 1563–1577, 1999.

[11] Y. Lu, A. Saxena, and T. F. Abdelzaher, "Differentiated caching services: A control-theoretic approach," in *Proc. Int. Conf. Distrib. Comput. Syst.*, Apr. 2001, pp. 615–654.

[12] S. Parekh, K. Rose, J. L. Hellerstein, S. Lightstone, M. Huras, and V. Chang, "Managing the performance impact of administrative utilities," in *Proc. IFIP Conf. Distrib. Syst. Oper. Manage.*, 2003, pp. 130–142.

[13] L. Sha, X. Liu, Y. Lu, and T. Abdelzaher, "Queueing model based network server performance control," in *Proc. IEEE Real-Time Syst. Symp.*, Dec. 2002, pp. 81–90.

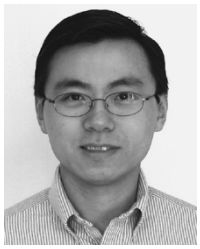
[14] S. Parekh, N. Gandhi, J. Hellerstein, D. Tilbury, J. Bigus, and T. S. Jayram, "Using control theory to acheive service level objectives in performance management," *Real-Time Syst. J.*, vol. 23, pp. 127–141, 2002.

[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," *IEEE/IFIP Netw. Oper. Manage.*, pp. 219–234, Apr. 2002.

[16] Y. Diao, J. L. Hellerstein, and S. Parekh, "Optimizing quality of service using fuzzy control," *Distrib. Syst. Oper. Manage.*, pp. 42–53, 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 *Proc. Amer. Control Conf.*, Jun. 2004, pp. 2045–2050.

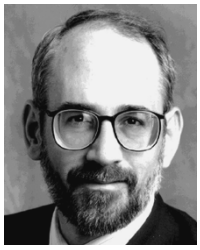
[18] Y. Diao, J. L. Hellerstein, G. Kaiser, S. Parekh, and D. Phung, "Self-managing systems: A control theory foundation," *Eng. Autonomic Syst.*, pp. 441–448, Apr. 2005.
[19] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury, *Feedback Control of Computing Systems.*. New York: Wiley, 2004.
[20] K. Ogata, *Modern Control Engineering*, 3rd ed. Englewood Cliffs, NJ: Prentice-Hall, 1997.
[21] IBM. (2004) Autonomic computing toolkit. Tech. Rep. [Online]. Available: http://www-106.ibm.com/developerworks/autonomic/probdet.html
[22] R. Zhang, C. Lu, T. F. Abdelzaher, and J. A. Stankovic, "Controlware: A middleware architecture for feedback control of software performance," in *Proc. Int. Conf. Distrib. Comput. Syst.*, 2002, pp. 301–310.
[23] TPC. (2004) (TPC) Web. Transaction Processing Council, Tech. Rep. [Online]. Available: http://www.tpc.org/tpcw



Yixin Diao (M'01) received the Ph.D. degree in electrical engineering from Ohio State University, Columbus, in 2000.

He is a Research Staff Member at the IBM Thomas J. Watson Research Center, Hawthorne, NY. He has published more than 30 papers and coauthored *Feedback Control of Computing Systems* (New York: Wiley, 2004). His research interests include systems management automation, adaptive control of dynamic systems, autonomic resource allocation, and modeling and optimization of distributed systems.

Dr. Diao is the recipient of several awards, including the 2002 Best Paper Award at the IEEE sponsored Network Operations and Management Conference and the 2002–2005 IFAC Theory Paper Prize in Engineering Applications of Artificial Intelligence.



Joseph L. Hellerstein (M'93–SM'98) received the Ph.D. degree in computer science from the University of California, Los Angeles.

He is a Research Staff Member and Manager of the Adaptive Systems Department, IBM Thomas J. Watson Research Center, Hawthorne, NY, and an Adjunct Professor at Columbia University, New York. He has authored or coauthored approximately 100 peer reviewed articles, an Addison-Wesley book on expert systems, and a book entitled *Feedback Control of Computing Systems* (New York, Wiley, 2004). His research has addressed various aspects of service quality in computing systems, including predictive detection, automated diagnosis, expert systems, and the application of control theory to computing systems.

Sujay Parekh (S'05) received the M.S. degree in computer science from the University of Washington, Seattle.

He is an Advisory Software Engineer at the IBM Thomas J. Watson Research Center, Hawthorne, NY. He is a coauthor of *Feedback Control of Computing Systems* (New York, Wiley, 2004). He has published several papers and contributed to autonomic features for IBM software. His research interests center around automating both simple and complex computing systems, and have included work in AI planning, machine learning, computer architecture, scheduling algorithms, and control systems.

Rean Griffith received the B.Sc. degree in computer science and management from the University of the West Indies, Barbados, in 2000 and the M.Sc. degree in computer science from Columbia University, New York, in 2003. Currently, he is working towards the Ph.D. degree in the Programming Systems Laboratory (PSL), Columbia University.

His research interests include adaptive systems, self-healing systems, and system manageability.

Gail E. Kaiser (M'85–SM'90) received the Sc.B. degree from the Massachusetts Institute of Technology, Cambridge, in 1979, and the M.S. and Ph.D. degrees from Carnegie Mellon University, Pittsburgh, PA, in 1980 and 1985, respectively.

She is a Professor of Computer Science and the Director of the Programming Systems Laboratory, Computer Science Department, Columbia University, New York. She has consulted or worked summers for courseware authoring, software process and networking startups, several defense contractors, the Software Engineering Institute, Bell Laboratories, IBM, Siemens, Sun, and Telcordia. Her laboratory has been funded by the Defense Advanced Research Projects Agency (DARPA), National Science Foundation (NSF), Office of Naval Research (ONR), National Aeronautics and Space Administration (NASA), NYS Science and Technology Foundation, and numerous companies. She served on the Committee of Examiners for the Educational Testing Service's Computer Science Advanced Test (the GRE CS test) for three years, and has chaired her department's doctoral program since 1997. She has published over 100 refereed papers in a range of software areas. Her research interests include self-managing systems (autonomic computing), publish/subscribe event systems, security, Web technologies, collaborative work, information management, distributed systems, and software development environments and tools.

Dr. Kaiser was named an NSF Presidential Young Investigator in Software Engineering and Software Systems in 1988. She served on the Editorial Board of *IEEE Internet Computing* for many years, and was a founding Associate Editor of *ACM Transactions on Software Engineering*. She Chaired an ACM SIGSOFT Symposium on Foundations of Software Engineering, Vice Chaired three of the IEEE International Conference on Distributed Computing Systems, and serves frequently on conference program committees.

Dan Phung received the B.S. degree in molecular biology from the University of New Mexico, Albuquerque, in 2001 and the M.S. degree from Columbia University, New York, in 2004. Currently, he is working towards the Ph.D. degree at Columbia University.

He has conducted research and published on a diverse range of topics such as human brain imaging, genomic mutations, and distributed software systems. His current research interests pertain to the management, dynamic resource allocation, and informed scheduling of high-performance systems.