Self-Managing Systems: A Control Theory Foundation

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

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, either from new components or layering on top of existing components. Further, 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, either from new components or layering on top of existing components.

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 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

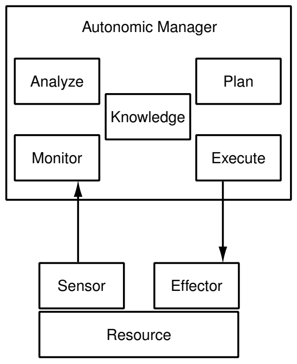

Figure 1. *Architecture for Autonomic Computing.*

be broadened to "enactment" to include manual actions as well.) Note scaling is achieved by having a single autonomic manager control multiple resources and by applying the architecture recursively so that lower level managers are treated as resources by higher level managers.

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 considered stochastic control [7]. Control theory has also been applied to middleware to provide service differentiation and regulation of resource utilizations (e.g., [8]), with considerations for nonlinearities [9] and including multiple-input, multiple-output control [10]. More recently, there has been impact on software products, such as the use of control theory to regulate administrative utilities in IBM's DB2 [11].

The foregoing illustrates the value of using control theory to construct self-managing systems. We take this a step further and explore control theory as a way to guide the development of autonomic systems. To this end, the paper has three goals. First, we seek to educate systems oriented computer science researchers and practitioners on the concepts and techniques needed to apply control theory to computing systems. Second, we describe how control theory can aid in building self-managing systems (possibly layered on top of legacy systems), and we identify the challenges in doing so. Last, we propose a deployable testbed for autonomic computing that is intended to foster research that addresses the challenges identified. The remainder of the paper is structured along the lines of these goals, with Section 2, Section 3, and Section 4 addressing each in turn. Our conclusions are contained in Section 5.

2 Control Theory Background

This section provides background on control theory and relates control theory to self-managing systems.

2.1 Components of a Control System

Over the last forty years, control theory has developed a fairly simple reference architecture (e.g., [12]). 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 knowledge.

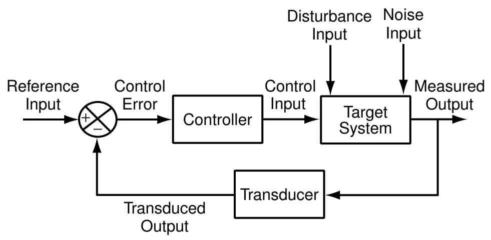

Figure 2. *Block diagram of a feedback control system.*

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

- 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.
- 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.
- 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 three 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 autonomic computing and control systems address similar concerns, there are some important differences as well. Autonomic computing has a strong emphasis on software architectures, their realization, and interoperation with legacy systems. For example, there is considerable focus on sensors and effectors so autonomic managers can control resources.

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, in a cluster of three web servers, the reference input might be that the utilization of a web server should be maintained at 66% to handle fail-over. If we add a fourth web server to the cluster, then we may want to change the reference input from 66% to 75%.

- 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 minimize 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 commercial information technology systems. Other properties may be of interest in different settings. For example, [13] discusses 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.

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,

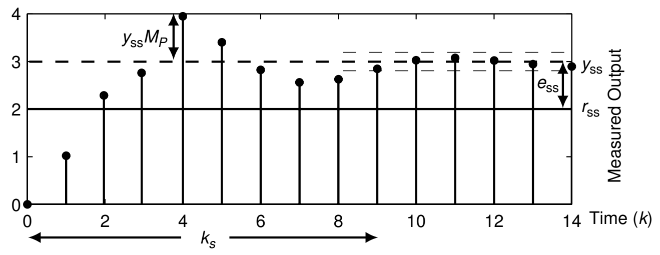

Figure 3. *Response of a stable system to a step change in the reference input. At time* 0*, the reference input changes from* 0 *to* 2*. The system reaches steady state when its output always lies between the light weight dashed lines. Depicted are the steady state error (e_{ss}), settling time (k_s), and maximum overshoot (M_P).*

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.

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 3 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 [14].

2.3 Control Analysis and Design

This subsection uses a running example to summarize an approach to control analysis and design.

We consider the IBM Lotus Domino Server based on work in [15]. 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. Based on these experiments, we constructed an empirical model (using least squares regression) that relates `MaxUsers` to RIS.

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

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

$$\frac{0.47}{z - 0.43}$$

The poles of a transfer function are the values of z for which the denominator is 0. It turns out that the poles determine the stability of the system represented by the transfer function, and they largely determine its settling time. This can be seen in Equation (1). Here, 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.

3 Building Self-Managing Systems with Control Theory

This section describes how control theory can be used to build self-managing systems (either from new components or layering on top of existing components), 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 [16].

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

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

This model works well in studies we have conducted [10].

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 [17]. This approach involves a detailed understanding of the TCP/IP protocol and the development of differential equations to estimate a transfer function.

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) [18]. 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 produce 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 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).

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 service level agreements 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 [19], 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., [20]). 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 [21]. 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, possibile resulting in stability problems. The latter arise because new control actions are taken without considering the impact of past actions, a situation that can result in an escalation of over-reactions.

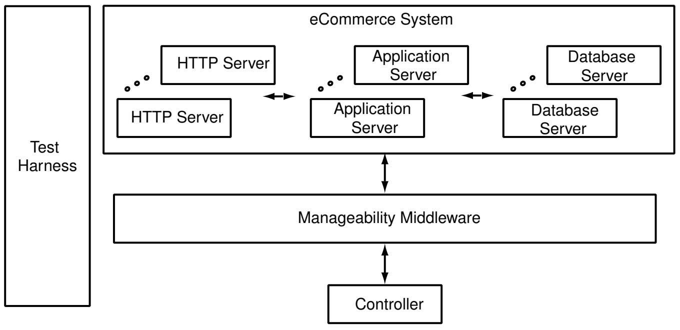

Figure 4. *Architecture of the Deployable Testbed for Autonomic Computing (DTAC).*

4 Deployable Testbed for Autonomic Computing (DTAC)

This section describes a deployable Testbed for autonomic computing. The Testbed addresses the following challenge in pursuing research in autonomic computing. Researchers tend to focus on one aspect of autonomic computing such as control techniques, management middleware, sensors, and effectors. However, to assess the value of work in any one area, a complete system must be developed, which is a significant effort.

The foregoing motivates 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 control algorithms implemented in the Testbed.

Figure 4 displays our initial architecture for DTAC. There are four layers in the architecture, all of which are intended to be pluggable. 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 obtaining 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 [22], and ControlWare [23]. 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). As such, this layer is the focal point for policy interpretation and enforcement. We are considering a couple of possibilities for controllers packaged with the Testbed including a PI (proportional integral) controller [24] and a fuzzy controller [25].

Last, the Test Harness provides the overall experimental environment, including the generation of synthetic workload, data collection, a suite of tools for analyzing experimental results, reporting and sending commands to all elements of the eCommerce System (e.g., for fault injection). Workload generation is of particular concern since it has a dramatic effect on the experimental results. Where possible, 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). However, we recognize that there is no standard benchmark for some aspects of autonomic computing, such as self-configuration and self-protection. Indeed, the development of such benchmarks is an important area of research.

We emphasize that Figure 4 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 PostgreSQL Database Server. All are publically available, both the executables and the source. Also, they are widely used in production systems.

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., [10]) 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) [26] since there is a publically available software driver.

5 Conclusions

This paper takes the position that control theory can provide an architectural and analytic foundation for building self-managing systems, either from new components or layering on top of existing components. 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 automation 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.

6 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. Corp., "An architectural blueprint for autonomic computing," Tech. Rep. http://www-306.ibm.com/autonomic/pdfs/ACwpFinal.pdf, 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 *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] T. F. Abdelzaher and N. Bhatti, "Adaptive content delivery for Web server QoS," in *International Workshop on Quality of Service*, (London, UK), June 1999.

[9] D. Henriksson, Y. Lu, and T. Abdelzaher, "Improved prediction for web server delay control," in *16th Euromicro Conference on Real-Time Systems*, (Catania, Italy), July 2004.

[10] 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.

[11] S. Parekh, K. Rose, Y. Diao, V. Chang, J. L. Hellerstein, S. Lightstone, and M. Huras, "Throttling utilities in the IBM DB2 universal database server," in *American Control Conference*, June 2004.

[12] R. Kalman, "On the General Theory of Optimal Control," in *Automatic and Remote Control*, pp. 481–492, 1961.

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

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

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

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

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

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

[23] 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.

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

[25] K. M. Passino and S. Yurkovich, *Fuzzy Control*. Addison Wesley Longman, Inc., 1998.

[26] TPC, "TPC Web," Tech. Rep. http://www.tpc.org/tpcw, Transaction Processing Council, 2004.