# Hierarchical Model-based Autonomic Control of Software Systems

Marin Litoiu
IBM Center for Advanced Studies
Markham, Canada
1-(905) 413-4095
marin@ca.ibm.com

Murray Woodside
Carleton University
Ottawa, Canada
1-(613) 520-5721
cmw@sce.carleton.ca

Tao Zheng
Carleton University
Ottawa, Canada
1-(613) 520-2600 x 5728
zhengtao@sce.carleton.ca

**ABSTRACT.** Various control algorithms are used in autonomic control to maintain Quality of Service (QoS) and Service Level Agreements (SLAs). Controllers are all based to some extent on models of the relationship between resources, QoS measures, and the workload imposed by the environment. This work discusses the range of algorithms with an emphasis on richer and more powerful models to describe non-linear performance relationships, and strong interactions among the system resources. A hierarchical framework is described which accommodates different scopes and timescales of control actions, and different control algorithms. The control algorithms and architectures can be considered in three stages: tuning, load balancing and provisioning. Different situations warrant different solutions, so this work shows how different control algorithms and architectures at the three stages can be combined to fit into different autonomic environments to meet QoS and SLAs across a large variety of workloads.

## Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures-*domain specific-architectures*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems- *distributed applications*; C.4 [**Computer-Communication Networks**]: Performance of Systems- *modeling techniques, performance attributes*; K.6.3 [**Management of Computing and Information Systems**]: Software Management-*software maintenance*;

## General Terms

Management, Measurement, Performance

## Keywords

Self-management, performance models, autonomic computing.

## 1. INTRODUCTION

Software systems can be adjusted to their users and environments in many ways, to provide satisfactory functionality and performance. If these adjustments are made at early stages in design they become limitations on the uses that can be made of the system in other environments. When they are made during deployment or installation they may limit adaptation to load levels. Autonomic systems solve this problem through built-in adaptive capability.

Autonomic systems have built-in points of dynamic variation that can be controlled at run time to modify their functionality or their use of resources. When they are installed, they first tune themselves to the demands of the users and the nature of the execution environment, and then they can track time-variations in the users and the environment, and accept updates to the system and its requirements. Roughly speaking, instead of making design and configuration choices they retain many choices, with algorithms for adapting the choices and a reflective architecture for gathering information, and for making the decisions and effecting the changes. In this, they imitate homeostatic systems found in nature.

Autonomic self-optimizing systems have two kinds of goals: they must maintain adequate quality of service (QoS), typically defined by service-level agreements (SLAs), and they may also seek to provide efficient operation, using a minimum of resources. The former goals are constraints to be maintained, such as constraints on average delay, or the probability of exceeding a target delay, and the latter goals provide a cost function to be minimized by tuning. Typically priority is given to maintaining the SLA constraints at all times, while seeking the minimum cost.

## 1.1 Autonomic Self-Optimizing Systems

Basic autonomic capability is provided by a controller with functions

- *monitor* (some system variables, denoted ***y***)
- *decide* (to make a change in a controlled parameter of the application or the operating system)
- *execute* (implement the decision, as control variables ***u***)

in a feedback configuration as in Figure 1. The Figure is based on the MAPE-K architecture for autonomic systems [18].

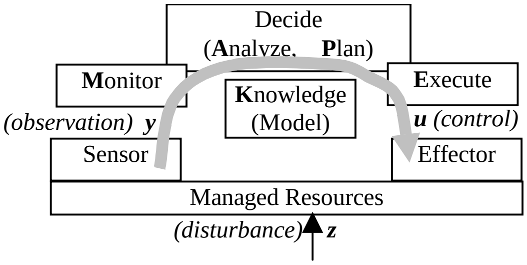

**Figure 1. Feedback control for autonomic systems**

The Decision operations determine the control, and may include

DEAS 2005, May 21, 2005, St. Louis, Missouri, USA.
Copyright 2005 ACM 1-59593-039-6/05/0005...$5.00.

heuristic estimators and controllers, or may be based on a system model (as in conventional linear system control [9]).

*Threshold control:* A common heuristic controller is a threshold adjustment function as illustrated in Figure 2. For example, in a web server the queue of requests is observed and when it passes a threshold, new threads are created; when it is zero for some time, idle threads are destroyed. The threshold reduces the frequency of control actions when the system is in a desirable regime, to reduce the cost of control.

*Policy Function control:* Threshold control has been generalized to multiple variables and control levels in [1], [6], [17] using an optimal control policy in an n-dimensional space, illustrated in Figure 3. The policy can be optimized by methods of Markov Decision Theory based on a Markovian model for the system behaviour.

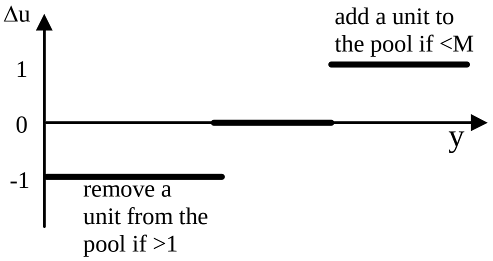

**Figure 2. Illustration of threshold control of a single resource**

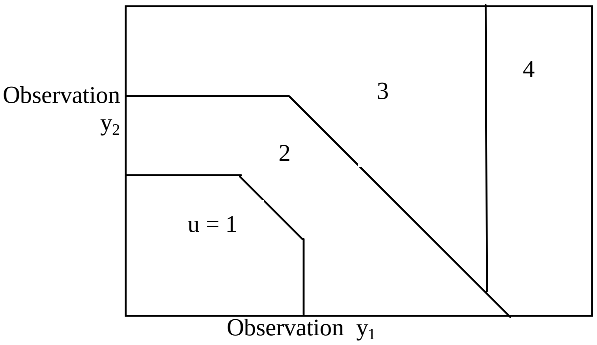

**Figure 3. Optimal u given as a control policy (as in [1])**

*Linearized dynamic control:* Hellerstein and his co-workers[4] and others have used controllers based on a linearized dynamic model given by equations such as:

$$x(k+1)=Ax(k)+Bu(k) \qquad (1)$$

$$y(k) = C\,x(k) + D\,u(k) + E\,z(k) \qquad (2)$$

with control equations based on a well-developed theory [2]. If the disturbance term *z* includes errors of observation then the optimal controller includes a filter to estimate the state from observations *y*. In [8] they controlled the memory used by a Lotus Notes application. Linearized dynamic control has been also used by Abdelzaher and co-workers [12] to adjust the number of threads in a web server.

In equations (1) and (2), *x* denotes the state variable vector of the dynamic model. The coefficient matrices *A B C D E* can be fitted to historical data [9], as a kind of regression model. They can be updated to give a tracking model, in case the system changes. The controlled parameter *u* is recomputed at every step.

The theory of control based on tracking models in a more general context was studied intensively by Astrom and co-workers (see [3]), and was applied to time-varying linear systems and to non-linear systems in which the operating point changed (requiring a new linearization). It is worth noting that most performance relationships are markedly non-linear, so linear models like this are not robust and need to be updated constantly. Updating can be quite slow.

The approaches in these three groups often do not monitor and control QoS directly. They apply indirect control through other variables such as utilization or queue length (see [2] for example). Thus their relationship to the SLA is indirect, and has to be calibrated. The secondary goal of optimization of resource use is often absent. The above approaches also control just one component of the software system and the interaction between different components or applications is ignored.

This paper proposes a framework which accommodates the above approaches, and also a more ambitious approach based on a holistic performance model for the interactions of the resources and workloads in the system. The model can be used to accelerate the autonomic reactions to workload changes, and to search for resource-optimal configurations. The types of systems that can benefit from our approach are *information- and transaction-based software* applications, such as e-commerce, insurance claim submission, Web banking, brokerage, and others. In these systems, users log-in, alternates requests with *think times* and then log-out. In terms of performance modeling, these systems are best described by closed models.

## 2. A HIERARCHICAL ARCHITECTURE

This paper describes a flexible approach to building autonomic systems with many control points (controlling points of variation) that have interacting effects. The approach is hierarchical, to separate controls with different localities and time-scales of application, and it can support different kinds of control policies, and different kinds of models of the system. Feedback policies to tune the control values can be combined with model-based accelerators to provide faster reaction to major changes.

Figure 4 shows the building blocks and the data flow of the proposed control architecture. The basic building blocks of the system, both for functionality and for management, comprise a set of Managed Components. Each managed component includes a management layer with sensors for monitoring and effectors for modifying controlled system parameters. To each Managed Component is attached a low-level autonomic controller, called the *Component Controller* or CController level. Information local to the component is used to adjust run-time tuning parameters, to achieve local QoS-related targets.

A model of the managed component is built and/or calibrated at runtime with the data provided by the sensors. The model helps the Component Manager to predict the effect of tuning one parameter or another. Examples of local tuning parameters are the size of a thread pool (which is often adjustable in, for example, web servers) and the size of a buffer pool or cache. Component tuning can be active at all times and is relatively fast and inexpensive, so it is the preferred way to achieve QoS goals. However it may not be capable by itself of maintaining a SLA, and it cannot set its own QoS targets.

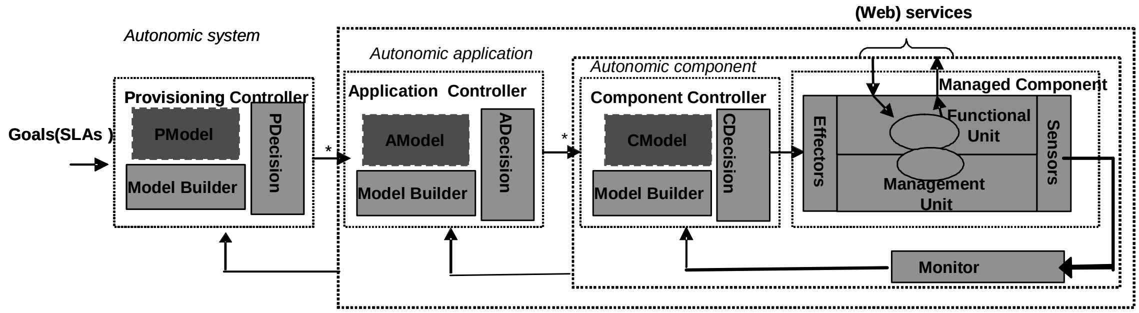

**Figure 4. Component diagram of hierarchical control of software systems**

The middle level is the *Application Controller* or AController level, which manages system-wide tuning and the interactions between the components. It sets QoS targets for the components, and takes over when they are incapable of maintaining their targets. Similarly to the Component Controller, the Application Controller builds a model of the system under control in order to evaluate and predict the effect of its load balance decisions. However, the model and the decisions are different than those for tuning as they have to take into account more component interactions. AController balances resources and workload across the system, involving interactions between components. Examples include the partitioning of request streams across a number of web servers, or replicated databases, the establishment of connection pools between processes, or the sharing of database buffer pools between application classes.

If the workload intensity changes substantially it may provide an opportunity for cost savings that can only be achieved by removing a node and its software from the active resources of the application. Similarly an increase of workload intensity may be still within the SLA and require that additional hardware and software components must be provided. *Self provisioning* is the ability of a system to add or remove instances of its hardware and software components at run-time without disrupting the application, and is provided by the third (highest) level, Provisioning Controller, or PController. Examples include adding and removing new web servers or application servers in a cluster.

Similarly to previous control levels, the Provision Controller builds a model of the system and evaluates different provisioning decisions before performing any change.

## 2.1 Architecture Implications

Figure 5 shows a framework which implements the hierarchical control model proposed in this paper. PController, AController and CController extend the class Controller which provides the basic interface. Controller interacts with a managed element at any level through Sensor and Effector interfaces, and bases its decisions on one or more Performance Models. The models, which are detailed later, can be grouped in four categories: Threshold, Policy, Dynamic and Queuing Models (which include layered queueing models (LQMs) as a special case).

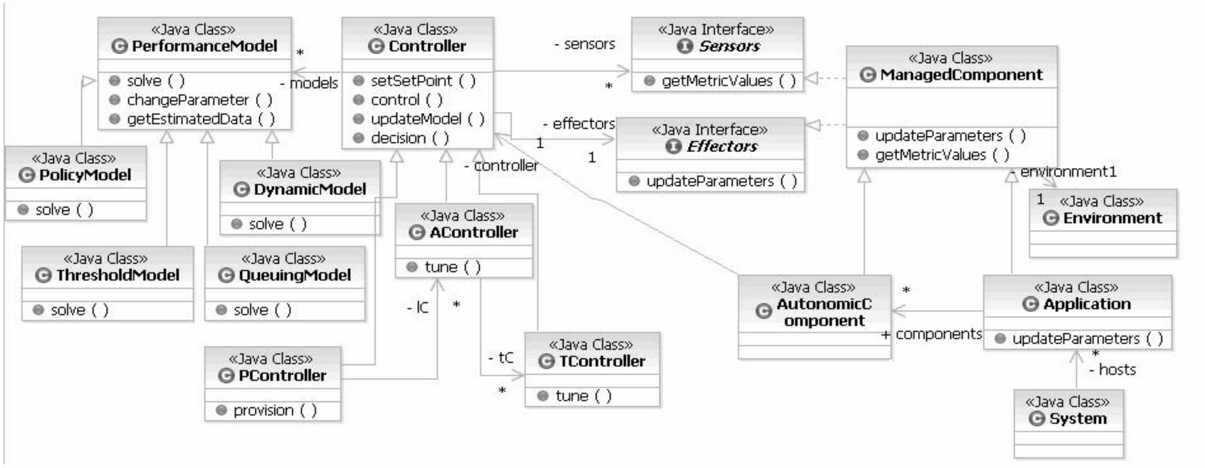

**Figure 5. Class diagram of an hierarchical self-management infrastructure**

A strength of the framework is the ability to change the model or control algorithm modules. Controller provides an interface which can be tailored to different kinds of algorithms and the data they require.

Since the models and the control decisions are at the core of the architecture, we will detail them in more details in the next sections.

## 2.2 Control Decisions by Level

The decisions about changes in controlled parameters are made by Decision elements at each level. A Decision element(xDecision in Figure 4 and decision() method in Figure 5) is based on an implicit or explicit model of the system, which characterizes the impact of the decision. The processing of a decision element can be summarized as follows:

1. monitor the managed component performance metrics and the input workload and setpoints
2. use the performance model to estimate future metrics and future adjustments of controlled parameters
3. if the future workloads cannot be accommodated by local adjustments, alert the upper level; otherwise perform the local adjustments.

A decision element at a higher level establishes setpoints for the lower level and it also has the ability to preempt and validate the changes at the lower levels.

As described above, the controllers at different levels differ in the types of changes they control. These differences are partly in scope (restricted to one component, or system-wide) and partly in time-scale and cost (changing the provisioning is the slowest and most expensive). A PController makes more dramatic and last resort changes in the system. It adds and removes servers and software components to the system to face the variations of the load that cannot be properly handled by the CController and AController. To implement those changes, a PController can use predefined workflows written in scripting languages. For example, user controlled provisioning in [10] uses a version of Python as a scripting language for defining provisioning workflows.

To perform component tuning, application tuning and self-provisioning, a software system has to be aware of its performance characteristics, to detect and predict changes in workload and to decide which action or combination of actions it should take. All management structures and actions can be viewed as being based on models of the system and its behaviour. For the simplest cases, the model may not be explicit, but to manage a complex system with interacting measures and decisions, a model is essential. Different kinds of model can be used; the choice of model and of control algorithm have implications for the system architecture. *This paper places the control strategy and the architectural choices together in the context of the system complexity and the required speed of adaptation to change.*

A standard reflective architecture is suitable for systems which are simpler in some sense (typically, for control of a single variable from a single measure), while a system-wide multivariate control architecture is required in more complex cases. The model may be incorporated directly or it may be solved or optimized, and the solutions incorporated as controllers.

## 2.3 Performance Models for Autonomic Controllers

Four kinds of performance models can be used at each level of control described above. The type of performance model has an influence on the decision elements and on the Controllers in general.

**A.** *Queue-based Performance Models* can predict the QoS measures from the resource levels and the values of environmental “disturbance” quantities. The model can be created statically from the plans for the system, or calibrated periodically with special measurements, or updated periodically from operational data. Then a search technique can be applied to the model to find resource levels or alternative configurations that will satisfy the QoS requirements (see Figure 6).

These performance models include queueing models and layered queues (LQMs). An example of combining LQMs and search for the optimal configurations is [11]. Most models that are practical to solve give steady-state performance measures. They handle system non-linearities, but to track changes in workload or environment parameters the model must be periodically updated. Fluid models can be used for systems in overload and provide rather simple dynamic models for these situations.

*Scope:* Menasce described an architecture with a component that manages its own resources using a queuing model [13][14]. LQMs can be used at any control level since it can be used to model systems, applications and components. We will detail the use of LQMs later in the paper.

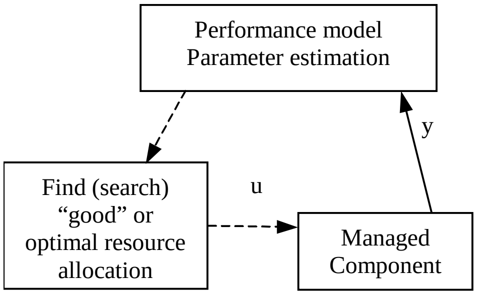

**Figure 6. Performance model-based resource management**

**B.** *Dynamic models,* such as those described in equations (1) and (2) in Section 1 can be used in a control structure shown in Figure 7. Once a model is built ( on- or off-line) a controller is designed. The controller (a linear function between u and y) continuously updates u as a function of the difference between the measured value y and the setpoint v. There are consecrated techniques for designing optimal controllers that provide a high quality of control while minimizing a cost function.

*Scope.* In principle any system can be modeled by the autoregressive schemes described in [8]. In a significantly nonlinear system the linear control may work badly or have to be updated frequently.

**C**. *Monotonic static models* relate a QoS measure to be controlled to a control value u, and give rise to threshold controllers. The controller is determined by discretising a performance model or else it expresses well-known heuristic principles in performance engineering. In terms of control style, the performance model and the control (decision) elements are combined in one entity.

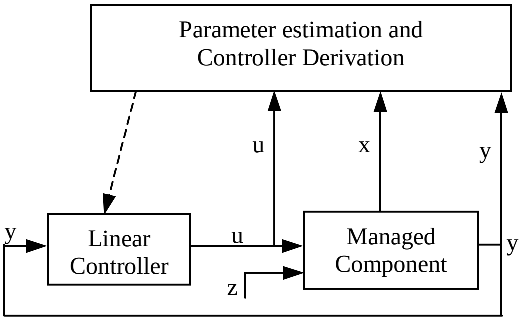

**Figure 7.** Adaptive dynamic control functions

*Scope:* Static models and threshold control are suitable for any of the levels of control detailed above, for a single controlled variable with negligible interactions with other resources. Reflective software components monitor a small number of parameters of their environment and behavior and modify themselves, if the measures go out of a desirable range. Such a reflective component can hide its adaptive behaviour, or it can export its measures and actions to an encapsulating self tuning manager. Performance-related autonomic behaviour can be provided if the reflective components encapsulate resources (as for example in processes with thread pools that adapt to demand).

**D.** *Policy based models*. An optimal policy may control multiple resources, based on multiple measures. Optimization can be directly on measures taken from the system, (as in taking the shortest delay path in updating a routing table according to observed delays), or it can be based on a model. As is the case with Threshold models, Policy models can be discrete representations of more complex models. A Markov Decision Process model has been used [1] to compute optimal reactions to state changes, combining observations on the system (for example, an observation of the system state) with assumptions about the rate of changes of state that are expected in the future. In an MDP model the variables u, y, and z take a discrete set of values only (such as “high”, “medium” and “low”).

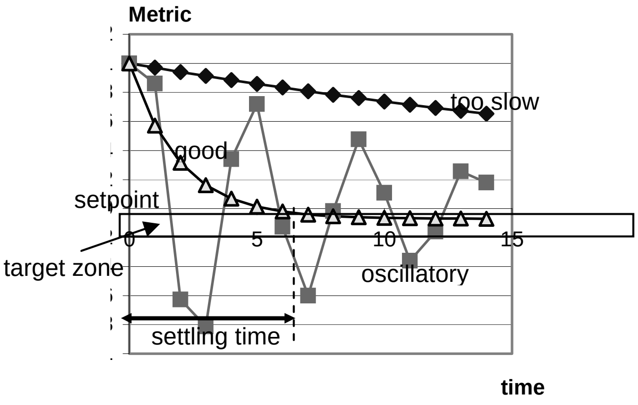

**Figure 8. The quality of control**

The optimal policy may be stored as a map, table or set of rules showing the policy to be applied in each situation, as a set of particular values of the control variables for that situation. Thus it resembles a decision table. An illustration as a map is given in Figure 3.

*Scope:* An optimal policy for one measure and one control is typically a threshold policy. Optimal policies over several measures and controls requires a kind of system-level reflection, gathering measures from several sources and controlling resources in a coordinated way across a sub-system.

## 2.4 Optimization and quality of control

As in any control system, good control brings the QoS into the desired range quickly, without either too-long delay or excessive oscillations. Figure 8 shows some possible responses over time of a controlled QoS metric related to delay, beginning from an unsatisfactory configuration with excessive delay. There is a target “setpoint” value and a target range including values near to but below the setpoint. The upper curve responds too slowly; the oscillating curve shows over-reaction; the middle curve settles into the desired range and stays there, within a reasonable “settling time”. What is achievable for a settling time depends on the time to obtain good estimates ad decisions, and the precision of decision making.

## 2.5 Accelerated Control

If an upper level controller is equipped with a sufficiently detailed model, such as a layered queueing model including the application and component resources, then it could use the model to determine optimal settings for its own level and those below it. This propagates suitable values for the lower-level controls to accompany a new value of the higher level control, and avoids the need for the lower level controllers to detect and adapt to the change on their own. If the model is only partly complete for the lower system, then the propagated values can cover what is included.

This acceleration is particularly effective in dealing with strong interactions between decisions at different levels.

The choice of which type of model or combination of models to use is driven by the nature of the system and the rapidity of changes in the “disturbances”. We’ll expand on the tradeoffs of each model in the next section

## 2.6 Discussion

The four control approaches above differ in: the nature and the scope of the model, in how often the models are built or recalibrated and how difficult is to build them, and in the optimization techniques that can be applied on top of them. We already described the scope of the models and how appropriate they are for each level of control and finally. In the remainder of this section we look at the rest of the differences.

*The nature of the model*

- A, C and D are static models, that is, they model the system in a steady or equilibrium state. B is a dynamic model, which can model transient phases of the system. As an example, A, C and D model and estimate well the performance of a software system when there is a constant number of users in the system, such as 100 or 200; B can estimate the performance metrics when the number of users suddenly changes from 100 to 200.
- C does not require an explicit model, and is limited to a local

scope (one measure determines one control). The others may be applied to any scope.

- C and D can use linear and non-linear models, B is a linear model, A models the non-linearities implicit in performance relationships.

*Model building and the speed of response to a changed situation*

Building the model of control in a timely manner and updating or calibrating it regularly is an important requirement for self-optimizing systems because, by definition, the system's structure and parameters are dynamically changed by the autonomic managers' actions.

- Models of type C are very component specific, they implement rules of thumb about the performance of the component and its internal resources. It is relatively easy to build, and responds very fast, but lose effectiveness and can have side effects if their monotonic assumptions are violated. For example, a simple model like, "if the number of users increases by x, increase the number of threads by 2x" become counterproductive when the memory is low or a resources become saturated.
- D covers more ground than C because it bases its decisions on more inputs. Rule-based control algorithms of type D can be effective as they take into account not only thresholds of the input variables, but the derivatives of those inputs. They also open the possibility of self-learning approaches to be used at runtime to dynamically change the parameters of the control algorithms.
- Model of type B are build in general experimentally by sampling the system across large number of input-output combinations. The relationships between x, y, u, z as well as well as the matrices A, B, C, D and E can hardly be extrapolated beyond the conditions under which sampled were taken. For example, the actions of load balancing or provisioning (such as adding or removing a server) can make the models of type B obsolete, when those models refer to system wide variables. Nevertheless, for systems that preserve their structure constant over large intervals, the models of type B can be very appropriate.
- Model of type A require measurement of resource consumption such as demand or service time with one user in the system. That poses a problem, since, at run-time, it is hard to accurately isolate one user request demand from the rest. Most of the research in building performance models of type A has concentrated so far on how to do it at development time. It is therefore conceivable that the models of type A are ready at the deployment time and then calibrated periodically at run time. An advantage of performance models is that structural changes in the system can be quickly reflected in the model. For example, adding and removing servers in the system is translated in the model in a change in the multiplicity of a cluster. Comparing with the rest of models, models of type A are easier to update; in the same time solving them takes longer.

## 3. THE USE OF LAYERED PERFORMANCE MODELS

Interacting components are a reality of many systems, including distributed systems with layered servers. Even within the same administrative scope there are components and resources that interact. The form of interaction is often through layered service patterns, and this work envisages the use of layered queueing models (LQMs) to represent an entire system or subsystem, to evaluate and optimize the use of resources. The high-level controllers will include the model and a technique for extracting control values or policies from it. Previous work by the authors has considered static optimization of resource use in layered systems. Litoiu and Rolia in [11] considered optimal allocation of processes to processors in a layered service system, and Zheng, El-Sayed, Cameron and Woodside in [5],[19] optimized priorities and allocations, to meet soft deadlines. Both of these works had good results with heuristic optimization strategies. Both used layered queueing models [7], [16],[20].

In real distributed systems it might not be enough to attempt to optimize a single component, such as a single application server (as in the strategy proposed in [12]). The bottleneck may move from one server to another, taking a long time to settle, and with incorrect threshold settings it might even thrash, with first one server being saturated (unloading a second server), such that an increase in capacity at the first server shifts the saturation to the other and unloads the first one. The best operating point may be many steps away from the current point. That is why we propose to execute the optimization search on a layered queueing model, in accelerated decision-making.

The use of layered performance models addresses several essential issues:

1. interactions between the usage of layered components, especially when congestion at a lower resource is creating saturation at higher resources (*software bottlenecks* [15])
2. the essential non-linearity in performance models as saturation is approached. This is a critical regime for correct decision-making for real systems.

The price that it requires is the creation of a hierarchical system-wide control architecture, which is less modular than the simple Q-components in [14]. This trade-off cannot be avoided. While completely decentralized decision-making (as in simple Q-components) may be able to eventually reach globally optimal operating points, we are convinced that it will take them much longer. We are studying this question.

If we can learn how to build a flexible overall architecture as described above, then the approach above would occupy one extreme (global control) and the simple Q-components, the other (every component providing its own control). The choice of the actual architecture used could be governed by the sharing of resources (as described above) and also by issues of practicality in obtaining and sharing information across the system.

## 4. CONCLUSIONS

Briefly,

- the hierarchical framework solves problems of scope and timescale differences in the adaptation;
- it also provides flexibility in choice of control algorithms, including migration over time if necessary,
- the framework has a distinct role for models of behaviour, and can accommodate many different models, including performance models. We intend to explore the use of layered

models (LQMs),

- it supports accelerated decision making based on LQMs, that propagates the effect of a high-level change rapidly into the lower levels.

The value of better models is the subject of current research, in the context of this framework.

## 5. ACKNOWLEDGEMENTS

This research was supported by the IBM Center for Advanced Studies (CAS), Toronto, and by the Natural Sciences and Engineering Research Council of Canada.

## 6. REFERENCES

[1] Abdeen, M. and Woodside, C. M. Seeking Optimal Policies for Adaptive Distributed Computer Systems with Multiple Controls. *Proc. Third International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'02)*, Kanazawa, Japan, Sept. 2002.

[2] Abdelzaher, T., Shin, K.J and Bhatti, N., Performance Guarantees for Web Server End-Systems: A Control-Theoretical Approach. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 13, No. 1, Jan 2002.

[3] Åström, K.J. and Wittenmark B. *Adaptive Control.* 2nd edition. Addison-Wesley Publ Co, 1995.

[4] Diao, Y., Lui, X., Froehlich, S., Hellerstein, J.L., Parekh, S. and Sha, L. On-Line Response Time Optimization of An Apache Web Server. *International Workshop on Quality of Service*, 2003.

[5] El-Sayed, H. E., Cameron, D. and Woodside, C. M. Automation Support for Software Performance Engineering. *Proc Joint Int. Conf on Measurement and Modeling of Computer Systems (Sigmetrics 2001/ Performance 2001),* Cambridge, MA, June 16 - 20, 2001, pp 301-311.

[6] Franken, L.J.N. and Haverkort, B.R. Reconfiguring Distributed Systems using Markov-Decision Models. *Proc. Trends in Distributed Systems (TreDS'96)*, Oct. 1996, pp. 219-228.

[7] Franks G., Majumdar, S., Neilson, J., Petriu, D., Rolia, J. and Woodside C.M. Performance Analysis of Distributed Server Systems. *The Sixth International Conference on Software Quality (6ICSQ)*, Ottawa, Ontario, 1996, pp. 15-26.

[8] Gandhi, N., Hellerstein, J. L., Parekh, S. and Tilbury, D. M. Managing the Performance of Lotus Notes: A Control Theoretic Approach. *Proceedings of the Computer Measurement Group*, 2001.

[9] Hellerstein, J., Diao, Y., Parech, S., Tilbury, D. *Feedback Control of Computing Systems*, John Wiley &Sons, Inc., 2004.

[10] IBM Tivoli Intelligent Orchestrator, http://www-306.ibm.com/software/tivoli/products/intell-orch/, Jan 23, 2005.

[11] Litoiu, M. and Rolia, J. Object Allocation for Distributed Applications with Complex Workloads. *Lecture Note in Computer Science* 1786, Springer, 2000, pp 25-39.

[12] Lu, Y., Abdelzaher, T., Lu, C., Sha, L. and Liu, X. Feedback Control with Queueing-Theoretic Prediction for Relative Delay Guarantees in Web Servers. *Real-Time and Embedded Technology and Applications Symposium*, Toronto, Canada, May 2003.

[13] Menasce, D. A. and Bennani, M. On the Use of Performance Models to Design Self-Managing Computer Systems. *Proc. 2003 Computer Measurement Group Conference*, Dallas, TX, Dec. 7-12, 2003.

[14] Menasce, D. A. QoS-aware software components. *IEEE Internet Computing*, March/April 2004, Vol. 8, No. 2.

[15] Neilson, J.E., Woodside, C.M., Petriu, D.C. and Majumdar, S. Software Bottlenecking in Client-Server Systems and Rendez-vous Networks. *IEEE Trans. On Software Engineering*. Vol. 21, No. 9, September 1995, pp. 776-782.

[16] Rolia, J. A. and Sevcik, K. C. The Method of Layers. *IEEE Trans. on Software Engineering*. vol. 21, August 1995. no. 8, pp. 689-700.

[17] Shin, K. G., Krishna, C. M. and Lee, Y-H. Optimal Dynamic Control of Resources in a Distributed System. *IEEE Transactions on Software Engineering*. Vol. 15, No. 10, October 1989.

[18] Stojanovic, L., Schneider, J., Maedche, A., Libischer, S., Studer, R., Lumpp, T., Abecker, A., Breiter, G. and Dinger, J. The role of ontologies in autonomic computing systems. *IBM Systems Journal*, v. 43, n. 3, 2004.

[19] Zheng, T. and Woodside, C. M. Heuristic Optimization of Scheduling and Allocation for Distributed Systems with Soft Deadlines. *Lecture Notes in Computer Science*, Springer-Verlag, vol. LNCS 2794, 2003, pp 169-181.

[20] Woodside, M. Tutorial Introduction to Layered Modeling of Software Performance, http://www.sce.carleton.ca/rads/lqn/lqn-documentation/, April 2005.