Utility Functions in Autonomic Systems

William E. Walsh, Gerald Tesauro, Jeffrey O. Kephart, and Rajarshi Das
IBM T.J. Watson Research Center
19 Skyline Drive, Hawthorne, NY 10532
{wwalsh1, tesauro, kephart, rajarshi}@us.ibm.com

Abstract

Utility functions provide a natural and advantageous framework for achieving self-optimization in distributed autonomic computing systems. We present a distributed architecture, implemented in a realistic prototype data center, that demonstrates how utility functions can enable a collection of autonomic elements to continually optimize the use of computational resources in a dynamic, heterogeneous environment. Broadly, the architecture is a two-level structure of independent autonomic elements that supports flexibility, modularity, and self-management. Individual autonomic elements manage application resource usage to optimize local service-level utility functions, and a global Arbiter allocates resources among application environments based on resource-level utility functions obtained from the managers of the applications. We present empirical data that demonstrate the effectiveness of our utility function scheme in handling realistic, fluctuating Web-based transactional workloads running on a Linux cluster.

1 Introduction

Self-optimization is an essential capability of autonomic computing systems [11]. More precisely, an autonomic computing system must optimize its own behavior in accordance with high-level guidance from humans. But what form should this guidance take, and what mechanisms should the system employ to translate this guidance into low-level actions that achieve the desired optimization objective? The central tenet of this paper is that *utility functions* can provide a general, principled and pragmatic basis for self-optimization in autonomic computing systems.

Utility functions are well-known in the fields of economics [15] and artificial intelligence [20] as a form of preference specification. In the context of autonomic computing, utility functions provide the objective function for self-optimization, mapping each possible state of an entity (an autonomic system or component) into a real scalar value. Typically, the state can be described as a vector of attributes, each of which is either measured directly by or synthesized from sensor measurements. The value may be expressed in any suitable unit, with monetary units being the most typical. The utility function might be specified by a human administrator, derived from a contract, or derived from another utility function.

Given a utility function, the system or component must use an appropriate optimization technique in conjunction with a system model to determine the most valuable feasible state and the means for achieving it. Typically, these means may include tuning system parameters or reallocating resources—both of which are explored in this paper. Since conditions are constantly changing, the optimization ought to be performed recurrently.

Utility functions have very attractive theoretical properties, but their use in practical autonomic computing systems is just beginning to be explored [3, 10]. The major contribution of this paper is to show how utility functions expressed in high-level business terms, or service-level attributes, can be used to dynamically allocate resources in a realistic autonomic computing system. This represents an advance over the existing literature, which either requires an administrator to ascribe economic value directly to low-level system resources, or assumes simple standard mappings between resources and quality of service. Our scheme supports multiple heterogeneous services by encapsulating their differences at a local level and providing a uniform means of communicating resource needs to a resource arbiter. The form of communication is a resource-level utility function that is derived locally from the service-level utility function by optimization algorithms coupled with a model.

The remainder of the paper is organized as follows. In Section 2, we review related work and distinguish our own contributions from the existing literature. In Section 3, we describe how utility functions, embedded in a suitable architecture and coupled with appropriate optimization and modeling technologies, can be used to drive the control and allocation of resources in a large-scale data center. We also describe a working prototype autonomic data center that

employs this architecture and set of technologies. Then, in Section 4, present and discuss some experimental results in the prototype. Finally, in Section 5, we summarize our findings and discuss future research challenges.

2 Related Work

Some authors [8, 13, 14, 17] have considered policies for controlling networks and distributed computing systems that are based on situation-action rules, which specify exactly what to do in certain situations. Such *Action policies* require policy makers to be intimately familiar with low-level details of system function—a requirement that is incompatible with the long-term goal of elevating human administrators to a higher level of behavioral specification.

Goal policies are a higher-level form of behavioral specification that establish performance objectives, leaving the system to determine the actions required to achieve those objectives. Many authors have considered how to allocate and control computational resources to guarantee promised levels of QoS. A common approach to meeting QoS goals is to restrict requests whose required QoS cannot be met [25, 27]. Since goals provide only a binary classification into "desirable" and "undesirable" performance, some have considered softer measures, such as maximizing the probability of achieving goals [7, 16, 19] or minimizing the degree to which goals are not met [4].

Utility functions can serve as an even higher-level form of behavioral specification, as they allow one to indicate degrees of desirability for different levels of QoS, perhaps distinguished by different applications or user classes. Utility functions permit on-the-fly determination of a "best" feasible state, while goal policies place the system in any state that happens to be both feasible and acceptable, with no drive towards further improvement. Utility-based resource allocation is not a new concept for computing systems. In 1968, Sutherland [22] proposed a futures market in which users could bid for computer time based on their own utility functions. More recent work has also focused on utility-based approaches [5, 10, 18]. In all of these approaches, utility is specified directly in terms of resources. However, a truly autonomic computing system should not require administrators to ascribe value to low-level resources. Instead, they should be able to specify utility in terms of the service-level attributes that matter to them or their customers, such as end-to-end response time, latency, throughput, etc.

Some market-based approaches [12, 23, 26] allow applications to specify their utility directly for goods representing QoS guarantees. The market contains agents that provide these QoS guarantees and know how to transform demand for QoS into demand for actual resources, and the market mechanism determines the resource allocation. This approach works well in domains where standard mappings between resources and QoS can be established. However, in a real data center the service specifications and the mappings from resource to QoS can be arbitrarily complex and application-specific. In the next section, we present an architecture and method that can cope with the complexity that will typify data centers and other autonomic computing systems.

3 Control and Resource Allocation in a Data Center

In this section, we illustrate how utility functions may be used effectively in autonomic systems by means of a data center scenario. The data center manages numerous resources, including compute servers, database servers, storage devices, etc., and serves many different customers using multiple large-scale applications. We focus in particular on the dynamic allocation and management of the compute servers within the data center, although our general methodology applies to multiple, arbitrary resources. In Section 3.1 we describe the high-level architecture of the data center model, and in 3.2 we describe the architecture the Application Managers, which manage individual applications. In Section 3.3 we outline the details of the data center implementation in a realistic prototype system.

3.1 Data Center Architecture

The data center, illustrated in Figure 1, contains a number of logically separated *Application Environments*, each providing a distinct application service using a dedicated, but dynamically allocated, pool of resources of various types, such as application servers, databases, or even virtual resources such as logical partitions.[1] An Application Environment also has a router to direct workload to servers. Each Application Environment has a *service-level utility function* specifying the business value of providing a given level of service to users of the Application Environment. The utility function may reflect the payment/penalty terms of service-level agreements with customers, and may also incorporate additional considerations such as the value of maintaining the data center's reputation for providing good service. We assume the utility function is independent of that of other Application Environments, and that all utility functions share a common scale of valuation, such as a common currency. The utility function for environment i is of the form $U_i(\mathbf{S}_i, \mathbf{D}_i)$, where $\mathbf{S}_i$ is the service level space in i and $\mathbf{D}_i$ is the demand space in i. Both $\mathbf{S}_i$ and $\mathbf{D}_i$ are vectors that specify values for multiple user classes. $\mathbf{S}_i$ is particular to i, and can contain any viable service metrics (e.g.,

[1] The inclusion of servers inside the Application Environment is meant to indicate a logical relationship, not physical proximity.

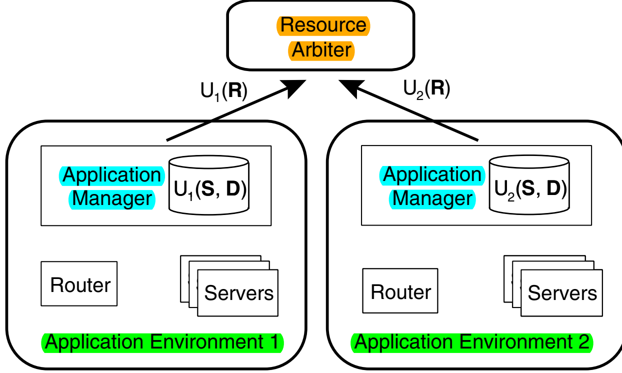

Figure 1. Data center architecture.

response time, throughput, etc.). Although such service-level specification of utility will often be most useful, we do not exclude the possibility that $\mathbf{S}_i$ could directly measure resources assigned to the classes in i.

The system goal is to optimize $\sum_i U_i(\mathbf{S}_i, \mathbf{D}_i)$ on a continual basis to accommodate fluctuations in demand. To this end, we present a distributed architecture, shown in Figure 1, consisting of multiple interacting autonomic elements. *Autonomic elements*, analogous to software agents, are the basic self-managing building blocks of autonomic computing systems. They manage their own behavior and their relationships with other autonomic elements, through which they provide or consume computational services [11]. The global optimization task is distributed among autonomic elements in a two-level structure.

At the lower level, the detailed control and optimization of a fixed amount of resources within an Application Environment is handled by a resident *Application Manager*. As demand shifts, Application Manager i may find it necessary to adjust certain control parameters or divert resources from one transaction class to another in order to keep $U_i(\mathbf{S}_i, \mathbf{D}_i)$ as optimal as possible, given a fixed amount $\mathbf{R}_i$ of resources. (Here $\mathbf{R}_i$ is a vector, each component of which indicates the amount of a specific type of resource that is allocated to Application Manager i.) Details of how the Application Manager optimizes $U_i(\mathbf{S}_i, \mathbf{D}_i)$ subject to fixed resource constraints are discussed in Section 3.2.

At the higher level, allocation of resources across different Application Environments is performed by a global *Resource Arbiter*. The Resource Arbiter is not privy to details of how the individual Application Managers optimize their utility, nor is it aware of any details of the services provided by the individual Application Environments. Instead, prompted by its own perceived need for more resource, or by a query from the Resource Arbiter, an Application Manager sends to the Arbiter a *resource-level utility function* $\hat{U}(\mathbf{R})$ that specifies the value to the Application Environment of obtaining each possible level $\mathbf{R}$ of resources.[2] Details of how the Application Manager computes $\hat{U}(\mathbf{R})$ from $U_i(\mathbf{S}_i, \mathbf{D}_i)$ are discussed in Section 3.2.

Given the current functions $\hat{U}_i(\mathbf{R}_i)$ from the Application Managers, the Resource Arbiter periodically recomputes the resource allocation $\mathbf{R}^*$ that maximizes the global utility $\sum_i U_i(\mathbf{S}_i, \mathbf{D}_i) = \sum_i \hat{U}_i(\mathbf{R}_i)$:

$$\mathbf{R}^* = \arg\max_{\mathbf{R}} \sum_i \hat{U}_i(\mathbf{R}_i) \text{ s.t. } \sum_i \mathbf{R}_i = \bar{\mathbf{R}}, \quad (1)$$

where $\bar{\mathbf{R}}$ indicates the total quantities of resources available. Eq. (1) is generally an NP-hard discrete resource allocation problem, and can be solved by a wide variety of standard optimization algorithms, including mixed-integer programming.

Our architecture is preferable to the more obvious centralized approach to global system optimization. It naturally supports the coexistence of multiple application environments that offer heterogeneous and arbitrarily complex services. The essential principle that enables this property is that each application environment is responsible for optimizing its own resource usage and for expressing its resource needs in a common, comparable form. All of the internal complexities of individual Application Environments, including representing and modeling a potentially infinite variety of services and systems, are compressed by the Application Manager into a uniform resource-level utility function that relates value to resources, all in common units. Our approach makes it easy to add, change or remove Application Environments—even different *types* of Application Environments—because the Resource Arbiter requires no information about their internal workings. Any reconfiguration required of other elements (e.g., making the Arbiter aware of a new Manager's existence) is handled automatically by the system, as described in greater detail by Chess et al. [6]. In contrast, a centralized approach would require constant updates to the Resource Arbiter.

Our two-level architecture also neatly handles the different time scales that are appropriate to different types of optimization by treating them independently. Application Managers adjust control parameters on a time scale of seconds to respond to changes in demand, while the Resource Arbiter typically operates on a time scale of minutes, which is more commensurate with switching delays necessitated by flushing out the current workload, changing connections, and installing or uninstalling applications.

3.2 Application Manager Architecture

In this section we describe the internal architecture of an Application Manager and show how it optimizes its utility

[2] If the resource-level utility function is sufficiently complex or expensive to compute, it is possible to avoid sending the entire function by having the Arbiter query each Application Manager for a limited set of $\mathbf{R}$ [2].

$U_i(\mathbf{S}, \mathbf{D}_i)$ subject to fixed resource constraints and computes $\hat{U}_i(\mathbf{R}_i)$ from $U_i(\mathbf{S}_i, \mathbf{D}_i)$. Since we restrict our attention to a single Application Manager here, we shall dispense with the i subscripts.

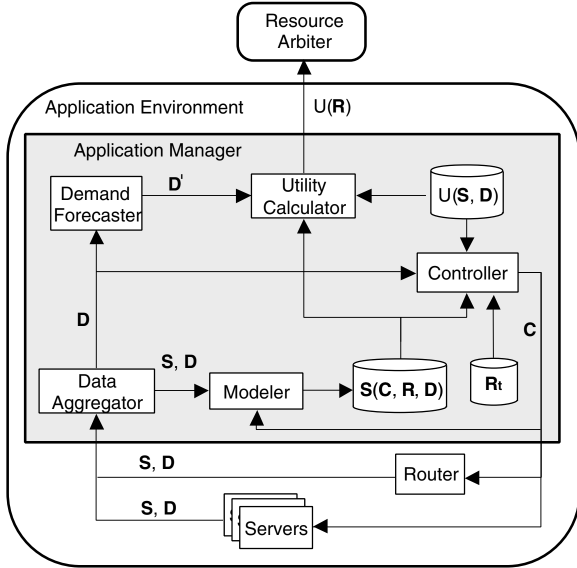

Figure 2. The modules and data flow in an Application Manager. Symbols: $\mathbf{S}$ = service level/service model, $\mathbf{D}$ = demand, $\mathbf{D}'$ = predicted demand, $\mathbf{C}$ = control parameters, $\mathbf{R}_t$ = current resource level, U = utility function.

Figure 2 illustrates the major components and information flows in an Application Manager. The Application Manager receives a continual stream of measured service $\mathbf{S}$ and demand $\mathbf{D}$ data from the router and servers. The *Data Aggregator* aggregates these raw measurements, e.g. by averaging them over a suitable time window. The *Controller* continually adjusts the router and server control parameters $\mathbf{C}$ in an effort to optimize the utility in the face of fluctuating demand. These parameters may specify how workloads from different customer classes are routed to the servers, as well as any other tunable parameters on the servers (e.g. buffer sizes, operating system settings, etc.).

The Application Manager maintains at least three kinds of knowledge: the service-level utility function $U(\mathbf{S}, \mathbf{D})$, the current resource level $\mathbf{R}_t$, and a model $\mathbf{S}(\mathbf{C}, \mathbf{R}, \mathbf{D})$ of system performance. The model specifies the vector of service levels that is obtained if the control parameters are set to $\mathbf{C}$, the resources allocated to the Application Environment is $\mathbf{R}$, and the demand is $\mathbf{D}$. The model yields a vector of expected service attribute measurements, which could, for example, represent one or more performance values for each customer class.

The Controller is responsible for optimizing the utility $U(\mathbf{S}, \mathbf{D})$ subject to fixed resource constraints. It receives the aggregated demand $\mathbf{D}$ from the Data Aggregator. When this quantity changes sufficiently, or other specified conditions occur, the Controller recomputes the control parameters $\mathbf{C}^*$ that optimize $U(\mathbf{S}, \mathbf{D})$ based on the performance model and current resource level:

$$\mathbf{C}^* = \arg\max_{\mathbf{C}} U(\mathbf{S}(\mathbf{C}, \mathbf{R}_t, \mathbf{D}), \mathbf{D}) \qquad (2)$$

and resets the control parameters to $\mathbf{C}^*$.

The *Utility Calculator* is responsible for computing the resource-level utility function $\hat{U}(\mathbf{R})$ from the service-level utility function $U(\mathbf{S}, \mathbf{D})$. Since shifting resources among different Application Environments may entail substantial delays, the Application Manager uses a *Demand Forecaster* to estimate the average future demand $\mathbf{D}'$ over an appropriate time window (e.g., up until the next reallocation), based on the historical observed demand $\mathbf{D}$ received from the Data Aggregator. The Demand Forecaster may use time series analysis methods, supplemented by special knowledge of the typical usage patterns of the application. The Utility Calculator computes the optimal resource-level utility $\hat{U}(\mathbf{R})$ that could be obtained based on the forecasted demand $\mathbf{D}'$. In other words, given the performance model $\mathbf{S}(\mathbf{C}, \mathbf{R}, \mathbf{D})$, and the service-level utility function $U(\mathbf{S}, \mathbf{D})$, the Utility Calculator computes

$$\hat{U}(\mathbf{R}) = \max_{\mathbf{C}} U(\mathbf{S}(\mathbf{C}, \mathbf{R}, \mathbf{D}'), \mathbf{D}') \qquad (3)$$

for all possible resource levels $\mathbf{R}$. Observe the similarity between Equations (2) and (3). To compute $\hat{U}(\mathbf{R})$ essentially requires repeated computation of (2) using each possible resource level $\mathbf{R}$, rather than just the current resource level $\mathbf{R}_t$, and with the predicted demand $\mathbf{D}'$, rather than the current demand $\mathbf{D}$.

With complex applications, it may be difficult for human developers to determine an accurate performance model *a priori*. To address this problem, the Application Manager can have a *Modeler* module that employs inference and learning algorithms to create, update, and revise the performance model based upon joint observations of $(\mathbf{S}, \mathbf{C}, \mathbf{R}_t, \mathbf{D})$.

3.3 The Prototype System

With our colleagues at IBM Research [6], we implemented a prototype of our data center in a general software architecture for autonomic systems called Unity. Unity provides a variety of autonomic elements written in Java using the Autonomic Manager ToolSet [1], and provides facilities for communication among elements so that different elements may run on different machines connected to a LAN or the Internet. The communication is based on standard OGSA [9] interfaces; other standard Web interfaces

are currently under development. We used Unity to cast the Resource Arbiter, the Application Managers, and individual servers as autonomic elements.

Unity also provides other autonomic elements not shown in Figure 1. These include a Registry, based on the Virtual Organization Registry [24], which enables elements to register and locate each other at run-time, and a Policy Repository, which stores the service-level utility functions for each Application Manager and provides an interface for modifying utility functions during run-time.

Unity has been implemented on a cluster of identical IBM eServer xSeries 335 machines running Redhat Enterprise Linux Advanced Server. The experimental results presented in Section 4.1 were generated by running two Application Managers and the Resource Arbiter on one machine; three other dedicated servers were made available as resources.

4 Examples of Utility-Based Allocation

To demonstrate the efficacy of our utility-function architectural scheme in a realistic system, we present examples of how it can be used to allocate and tune resources. First, in Section 4.1, we demonstrate resource allocation in the context of a data center prototype in which there are two Application Environments, each with a single transaction class. Each Environment has different service-level utility functions based on completely different metrics. The Application Managers respond to locally changing demand, and even changes in their service-level utility functions, by continually recomputing their resource-level utility functions, resulting in optimal dynamic resource allocation from the Resource Arbiter. Then, in Section 4.2, we illustrate how we would use utility functions to optimally tune resource parameters within a single Application Environment with multiple transaction classes; this approach has yet to be implemented within the prototype.

4.1 Single Transaction Class Per Environment

In this section we show examples of utility-based allocation run on the prototype Unity system with two Application Environments, each containing a single transaction class. Here, since **S**, **D** and **R** are all single-valued, we shall replace them with the scalar notation S, D, and R.

Application Environment A1 handles a transactional workload that provides a realistic simulation of an electronic trading platform. This transactional workload runs on top of WebSphere and DB2, both of which are installed on all the servers in Unity. The service-level utility function for A1 is defined solely in terms of the average response time S_1 of the customer requests. More precisely, its service-level utility function $U_1(S_1, D_1) = U_1(S_1)$ alternates between the two sigmoid functions shown in Figure 3b.

The customer demand for the transactional workload D_1 is generated by repeated requests for the login web page at a variable rate. Each time the page is accessed, the application retrieves the portfolio information of a randomly selected customer from its database, and displays the current values of the holdings, updated according to simulated market fluctuations. To provide for a realistic simulation of periodic and bursty web traffic, we use a time-series model developed by Squillante et al. [21] to reset the demand generated by the transactional workload every ~ 5 seconds.

Given the service-level utility function $U_1(S_1)$, A1 uses a simple system performance model $S_1(\mathbf{C}_1, R_1, D_1)$ to estimate the resource-level utility function $\hat{U}_1(R_1)$ for each possible number of servers R_1. Here, we hold the control parameters of the servers constant, allowing us to simplify $S_1(\mathbf{C}_1, R_1, D_1)$ to $S_1(R_1, D_1)$. (In Section 4.2 we shall reinstate $\mathbf{C}_1$, permitting us to adjust the relative resource consumption of two different transaction classes.) Prior to our main experiment, we obtained the performance model $S_1(R_1, D_1)$ (illustrated in Figure 3a) by measuring the average response time at each of several values of D_1 for 1, 2, and 3 servers. Each data point was sampled for 15 minutes, allowing us to average over several hundred to a few thousand transactions, and all non-sampled points were generated by linear interpolation. In these experiments, the Demand Forecaster simply returns the current demand.

Application Environment A2 handles a long-running batch workload. The service level S_2 is measured solely in terms of the number of servers R_2 allocated to A2. We took the utility function $U_2(S_2, D_2) = U_2(R_2) = \hat{U}_2(R_2)$ to be the piecewise linear function of R_2 shown in Figure 3c. Note that Figure 3c is on a different utility scale than Figure 3b, reflecting the lower value placed on the batch workload.

Results from a typical experiment with the two Application Environments and three servers are shown in Figure 4. The figure shows seven time-series plots over a period of 575 seconds. From top to bottom, they are: (**1**) Average demand D_1 on A1; (**2**) Average response time S_1 in A1; (**3**) Resource-level utility $\hat{U}_1(R_1)$ for $R_1 = \{1, 2, 3\}$ servers for A1; (**4**) Total utility from the two applications (solid plot) and the utility $U_1(S_1)$ obtained from A1 (dashed plot); (**5**) Utility $U_2(R_2)$ obtained from A2; (**6**) Number of servers R_1 allocated to A1; and (**7**) Number of servers R_2 allocated to A2. Notable times are indicated by vertical dashed lines and labeled by letters at the top.

Initially, we set $U_1(S_1)$ to be the relatively sharp sigmoid labeled U_1 in Figure 3b, with a transition from maximum to minimum utility centered around 30ms. The transaction rate D_1 begins low, allowing A1 to obtain a low response time and utility of nearly 1000 with one server; the other two servers are allocated to A2. At time **a**, D_1 rises, and the

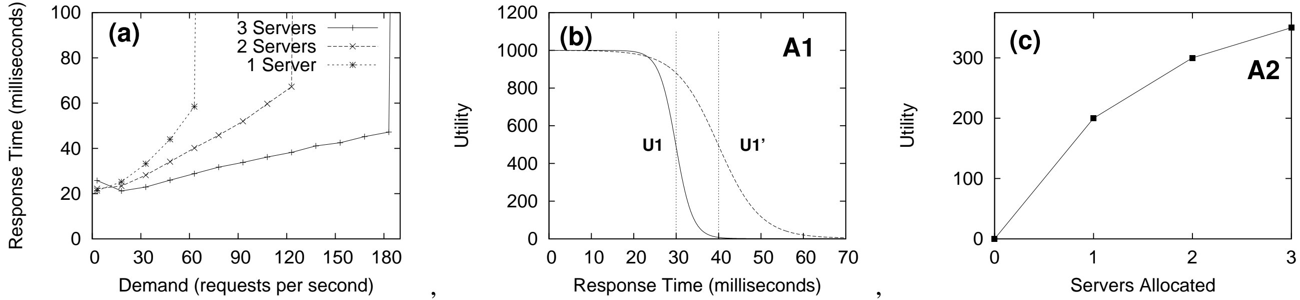

Figure 3. (a) The system model, (b) Service-level utility functions for Application Environment A1, and (c) Utility function for Application Environment A2.

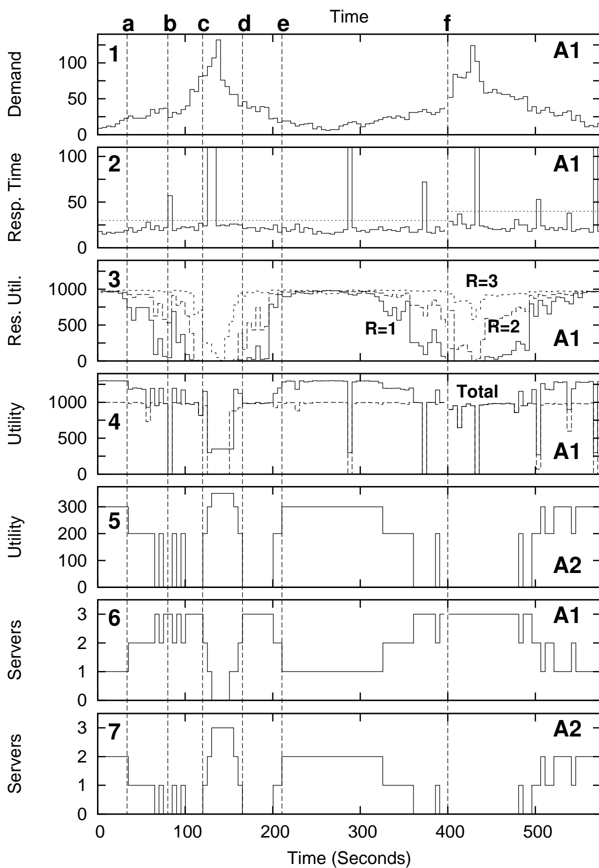

Figure 4. Times series plots, during a sample Unity experiment, of 1) D_1, 2) S_1, 3) $\hat{U}_1(R_1)$ 4) measured total utility U_1+U_2 (solid plot) and measured $U_1(S_1)$ (dashed plot), 5) $U_2(S_2,D_2)=\hat{U}_2(R_2)$, 6) R_1, and 7) R_2.

Manager of A1 changes $\hat{U}_1(R_1)$ so that two or more servers are needed to get a high utility. Upon receiving this utility information, the Arbiter computes a new optimal allocation, giving two servers to A1. Just after time **c**, the demand rises considerably, and even three servers are not enough to reduce S_1 enough to give A1 nonzero utility. Since no amount of the available servers can help A1, they are all allocated to A2. After the demand drops from its peak at **d**, $\hat{U}_1(R_1)$ for $R_1=3$ jumps to nearly 1000 and all three servers are transferred back to A1. As demand continues to decrease, the servers become less valuable to A1, hence the Arbiter reallocates the extra servers to A2, until time **e** at which A1 has only one server. Note that the spikes in response time for A1 at time **b** and just prior to 300 seconds, which are too transitory to trigger reallocation, are not caused by an increase in demand. Investigation reveals that they are due to Java garbage collection in WebSphere.

At time **f**, the policy repository interface is used to alter $U_1(S_1)$ to the curve labeled $U1'$ in Figure 3b, which has its transition centered at 40ms. This is communicated to the manager of A1, which determines that it can obtain high utility with fewer servers. Even at the demand peak after time **f**, the Manager of A1 computes $\hat{U}_1(R_1)$ to reflect that three servers are sufficient to give A1 positive utility.

Our experiments demonstrate that utility functions can form an effective and consistent basis for self-optimization in autonomic computing systems. Each Application Environment has a different utility function based on completely different measures. Nevertheless, our system automatically responds to changes in both demand and utility functions. Adding or removing Environments would result in similar autonomic behavior. The key is that most of the detailed knowledge and control complexity is managed by the individual Application Managers, while system-wide optimal behavior emerges from communication of common resource-level utility functions.

4.2 Multiple Transaction Classes In an Environment

Introducing multiple transaction classes in an Application Environment requires a more complex service-level

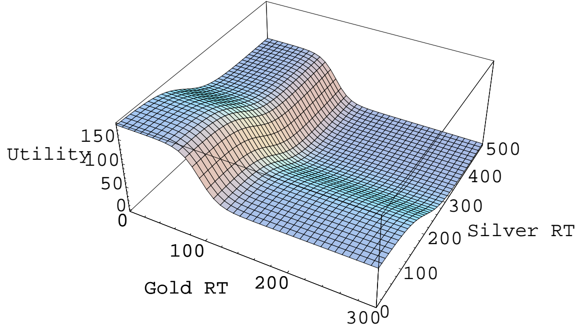

Figure 5. Utility as a function of joint response time for Gold and Silver transaction classes.

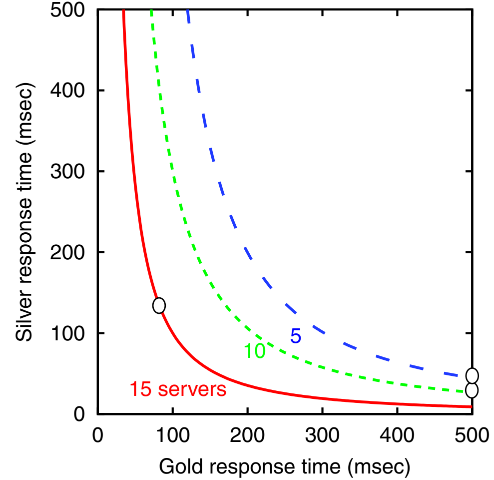

Figure 6. Feasible response times for different numbers of servers.

utility function and model, and also provides the opportunity for resource tuning within an Environment. Figure 5 shows an example service-level utility function, specifying value as a function of the joint response time for Gold and Silver transaction classes. In this case, we chose a function that is simply the sum of two independent utility functions for each class, each a function of the response time for its respective class only (again, for simplicity, we assume that the utility functions do not depend on demand).

To optimize $U(\mathbf{S},\mathbf{D})$ for multiple transaction classes, given current demand $\mathbf{D}$ for both transaction classes and fixed number of homogeneous servers, the Application Manager can adjust parameters in the router and servers to control the relative rate at which Gold and Silver transactions are processed (e.g., by decreasing the rate that transactions from one class are admitted relative to the other). Figure 6 shows a portion of a hypothetical performance model based on an M/M/1 queue that accounts for this control tradeoff. For numbers of servers $R \in \{5,10,15\}$, the associated hyperbolic curve is the boundary of $S(\mathbf{C},R,\mathbf{D})$, the feasible region for fixed number of servers R. The feasible regions are to the top and right of the curves. Any particular value $\mathbf{C}$ for the control parameters gives a point $\mathbf{S}$, specifying both Gold and Silver response time in the feasible space. Because $U(\mathbf{S},\mathbf{D})$ in Figure 5 specifies a preference for lower response times, the optimal utility for a fixed R must lie on the hyperbolic curve for R. Inspection of $U(\mathbf{S},\mathbf{D})$ will confirm that optimal, feasible service levels are indicated by empty dots on each curve. This optimization must be performed for all possible R to compute $\hat{U}(R)$.

5 Conclusions and Research Challenges

We have argued that utility functions provide a general and elegant basis for self-optimization in autonomic computing systems, and we have illustrated their practicality by means of a realistic data center prototype. Utility functions provide principled criteria for trading off between multiple competing system objectives. Moreover, used in conjunction with appropriate modeling and optimization algorithms, utility functions provide a basis for translating high-level business objectives based on service level considerations into lower level dynamic resource allocation decisions. Our distributed two-level architecture supports flexibility, modularity, and self-management in a heterogeneous system by encapsulating local complexity while still supporting global resource allocation. It does so by having each local environment express its resource needs to an arbiter in a uniform, comparable form. Although we demonstrated the principle in the context of a data center, we believe our approach will prove to be broadly useful in autonomic computing systems.

There is additional work to be done to allow our system to handle more of the complexities of real-world data centers. We must develop techniques to handle scaling in multiple dimensions, including: number of transaction classes, number of Application Environments, quantities of resource, and number of resource types. Scaling to multiple transaction classes or complex multi-tier resource configurations will increase the service model complexity. Clearly, table-based modeling derived from offline measurements (which we employed in Section 4) will be infeasible, and alternative modeling techniques will be necessary. Models based on queuing theory approximations or simulation will be required in the more complex environments, and furthermore these models will have to be learned and refined online, during the operation of the system. Thus the Modeler component of an Application Manager will require a fairly sophisticated learning component.

Another area for further investigation is how to account

for switching costs—the business value lost when resources are in the process of being reallocated between Application Environments. Switching costs are likely to introduce considerable complexity into the modeling and optimization procedures.

Finally, although utility functions are the natural way to represent value, humans will often find it difficult to express their utility for various components of a large, complex system. Carefully designed interfaces and preference elicitation techniques are needed to represent human notions of value accurately.

Overall, despite the significant work remaining to be done, we believe that our architecture is sufficiently general to accommodate most of the main innovations required to handle the above-mentioned complications.

References

[1] W. C. Arnold, D. W. Levine, and E. C. Snible. Autonomic manager toolkit. `http://dwdemos.dfw.ibm.com/actk/common/wstkdoc/amts`, 2003.

[2] C. Boutilier, R. Das, J. O. Kephart, G. Tesauro, and W. E. Walsh. Cooperative negotiation in autonomic systems using incremental utility elicitation. In *Nineteenth Conference on Uncertainty in Artificial Intelligence*, pages 89–97, 2003.

[3] A. Byde, M. Sallé, and C. Bartolini. Market-based resource allocation for utility data centers. Technical Report HPL-2003-188, HP Laboratories Bristol, Sept. 2003.

[4] A. Chandra, W. Gong, and P. Shenoy. Dynamic resource allocation for shared data centers using online measurements. In *International Workshop on Quality of Service*, pages 381–400, 2003.

[5] J. S. Chase, D. C. Anderson, P. N. Thakar, and A. M. Vahdat. Managing energy and server resources in hosting centers. In *18th Symposium on Operating Systems Principles*, 2001.

[6] D. Chess, A. Segal, I. Whalley, and S. White. Unity: Experiences with a prototype autonomic computing system. In *International Conference on Autonomic Computing*, 2004.

[7] R. P. Doyle, J. S. Chase, O. M. Asad, W. Jin, and A. M. Vahdat. Model-based resource provisioning in a web service utility. In *Fourth USENIX Symposium on Internet Technologies and Systems*, 2003.

[8] C. Efstratiou, A. Friday, N. Davies, and K. Cheverst. Utilising the event calculus for policy driven adaptation on mobile systems. In *3rd International Workshop on Policies for Distributed Systems and Networks*, pages 13–24, 2002.

[9] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the Grid: An Open Grid Services Architecture for distributed systems integration. Technical report, Open Grid Services Architecture WG, Global Grid Forum, `https://forge.gridforum.org/projects/ogsa-wg`, 2002.

[10] T. Kelly. Utility-directed allocation. In *First Workshop on Algorithms and Architectures for Self-Managing Systems*, 2003.

[11] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–52, 2003.

[12] S. Lalis, C. Nikolaou, D. Papadakis, and M. Marazakis. Market-driven service allocation in a QoS-capbable environment. In *First International Conference on Information and Computation Economies*, 1998.

[13] H. Lutfiyya, G. Molenkamp, M. Katchabaw, and M. Bauer. Issues in managing soft QoS requirements in distributed systems using a policy-based framework. In *2nd International Workshop on Policies for Distributed Systems and Networks*, 2001.

[14] L. Lymberopoulos, E. Lupu, and M. Sloman. An adaptive policy based management framework for differentiated services networks. In *3rd International Workshop on Policies for Distributed Systems and Networks*, pages 147–158, 2002.

[15] A. Mas-Colell, M. D. Whinston, and J. R. Green. *Microeconomic Theory*. Oxford University Press, 1995.

[16] S. Pampal, D. S. Reeves, and I. Viniotis. Dynamic resource allocation based on measured QoS. Technical Report TR 96-2, North Carolina State University, 1996.

[17] A. Ponnappan, L. Yang, and R. Pillai.R. A policy based QoS management system for the IntServ/DiffServ based internet. In *3rd International Workshop on Policies for Distributed Systems and Networks*, pages 159–168, 2002.

[18] R. Rajkumar, C. Lee, J. P. Lehoczky, and D. P. Siewiorek. Practical solutions for QoS-based resource allocation problems. In *IEEE Real-Time Systems Symposium*, pages 296–306, 1998.

[19] J. Rolia, X. Zhu, M. Arlitt, and A. Andrzejak. Statistical service assurances for applications in utility grid environments. In *Tenth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 247–256, 2002.

[20] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, second edition, 2003.

[21] M. S. Squillante, D. D. Yao, and L. Zhang. Internet traffic: Periodicity, tail behavior and performance implications. In *System Performance Evaluation: Methodologies and Applications*, 1999.

[22] I. E. Sutherland. A futures market in computer time. *Communications of the ACM*, 11(6):449–451, 1968.

[23] P. Thomas, D. Teneketzis, and J. K. MacKie-Mason. A market-based approach to optimal resource allocation in integrated-services connection-oriented networks. *Operations Research*, 50(4), 2002.

[24] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maguire, T. Sandholm, P. Vanderbilt, and D. Snelling. Open Grid Services Infrastructure (OGSI) version 1.0. Technical report, Open Grid Services Infrastructure WG, Global Grid Forum, `https://forge.gridforum.org/projects/ogsi-wg`, 2002.

[25] S. Wang, D. Xuan, R. Bettati, and W. Zhao. Providing absolute differentiated services for real-time applications in static-priority scheduling networks. In *IEEE Infocom*, 2001.

[26] H. Yamaki, M. P. Wellman, and T. Ishida. A market-based approach to allocating QoS for multimedia applications. In *Second International Conference on Multi-Agent Systems*, pages 385–392, 1996.

[27] J. Yoon and R. Bettati. A three-pass establishment protocol for real-time multiparty communication. Technical Report 97-006, Texas A&M University, 1997.