RC23357 (W0410-015) October 1, 2004
Computer Science

IBM Research Report

A Multi-Agent Systems Approach to Autonomic Computing

Gerald Tesauro, David M. Chess, William E. Walsh, Rajarshi Das, Alla Segal, Ian Whalley, Jeffrey O. Kephart, Steve R. White
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home.

A Multi-Agent Systems Approach to Autonomic Computing

Gerald Tesauro, David M. Chess, William E. Walsh, Rajarshi Das,
Alla Segal, Ian Whalley, Jeffrey O. Kephart and Steve R. White
IBM T.J. Watson Research Center
19 Skyline Drive, Hawthorne, NY 10532
{gtesauro,chess,wwalsh1,rajarshi,segal,inw,kephart,srwhite}@us.ibm.com

Abstract

The goal of autonomic computing is to create computing systems capable of managing themselves to a far greater extent than they do today. This paper presents Unity, a decentralized architecture for autonomic computing based on multiple interacting agents called autonomic elements. *We illustrate how the Unity architecture realizes a number of desired autonomic system behaviors including goal-driven self-assembly, self-healing, and real-time self-optimization. We then present a realistic prototype implementation, showing how a collection of Unity elements self-assembles, recovers from certain classes of faults, and manages the use of computational resources (e.g. servers) in a dynamic multi-application environment. In Unity, an autonomic element within each application environment computes a resource-level utility function based on information specified in that application's service-level utility function. Resource-level utility functions from multiple application environments are sent to a Resource Arbiter element, which computes a globally optimal allocation of servers across the applications. We present illustrative empirical data showing the behavior of our implemented system in handling realistic Web-based transactional workloads running on a Linux cluster.*

1. Introduction

The vision of autonomic computing [8] is of computing systems that manage themselves to a far greater extent than they do today. To achieve this vision, we believe that interacting sets of individual computing elements must regulate and adapt their own behavior in response to widely changing conditions, with only high-level direction from humans.

While traditional approaches to computer systems management are often centralized and hierarchical, today's large-scale computing systems are highly distributed with increasingly complex connectivity and interactions, rendering centralized management schemes infeasible. We propose instead that a multi-agent systems (MAS) approach is much better suited for autonomic computing. Jennings [6] advocates an agent-based approach to software engineering based on decomposing problems in terms of decentralized, autonomous agents that can engage in flexible, high-level interactions. This approach is particularly well-suited for autonomic computing systems, which must self-configure, self-protect, self-heal, and self-optimize on both local and system levels. Many ideas developed in the MAS community, such as those pertaining to automatic group formation, emergent behavior, multiagent adaptation, and agent coordination, among others, could likely be fruitfully adapted for autonomic computing. The practical challenges of automatic computing may likewise spur basic research advances within the MAS community.

Motivated by the above considerations, we have developed a software architecture called Unity, which aims to achieve self-management of a distributed computing system via interactions amongst a population of autonomous agents called autonomic elements. The specific software engineering objectives of Unity are: to enable a distributed system to self-configure at runtime initialization; to develop software design patterns that enable self-healing within the system; and to self-optimize in real time the use of distributed computational resources in accordance with the system's overall business objectives. We have developed a prototype data center in Unity as a concrete testbed to pursue these objectives. The data center, based on industrial systems under research and development, provides a number of computational resources to run multiple applications for many customers. Managing data centers is of real and practical concern for corporate information technology and provides many interesting challenges for autonomic computing.

Unity performs self-optimization based on utility functions that express the value to the data center as a function of the service level attained for customers and other users. The MAS architecture encapsulates the local optimization of resource usage from the global allocation of resources.

A key contribution of this paper is a working demonstration of service-level utility functions operating in the context of our prototype data center.

The remainder of the paper is organized as follows. We begin by outlining the basic structure and components of the Unity system in Sections 2. We then describe Unity's goal-driven self-assembly methodology in Section 3, and its design for cluster self-healing in Section 4. We then present our scenario for optimal resource allocation in a dynamic multi-workload environment in Section 5. Implementation details and empirical observations of our prototype system are given in Section 6, following by concluding remarks and future research directions in Section 7.

2. The Structure of Unity

The essential structure of Unity follows that outlined by Kephart and Chess [8]. The Unity system components are implemented as *autonomic elements*— individual agents that control resources and deliver services to humans and to other autonomic elements. Every Unity component is an autonomic element. This includes: computing resources (e.g. databases, storage systems, servers), higher-level elements with some management authority (e.g. workload managers or provisioners), and elements that assist other elements in doing their tasks (e.g. policy repositories, sentinels, brokers, or registries). We are particularly interested in the properties that all autonomic element subtypes have in common.

Each autonomic element is responsible for its own internal autonomic behavior, namely, managing the resources that it controls, and managing its own internal operations, including self-configuration, self-optimization, self-protection, and self-healing. Each element is also responsible for forming and managing relationships that it enters into in order to accomplish its goals, that is, the external autonomic behavior that enables the system as a whole to be self-managing.

Unity's autonomic elements are implemented as Java programs, using the Autonomic Manager Toolset [1] . They communicate with each other using a variety of Web Service interfaces, including both standard OGSA [5] interfaces and additional interfaces that we and others have defined for autonomic elements. An important principle of the system is that no other means of communication between the elements is permitted; there are no back doors or undocumented interfaces between elements. This allows us to completely specify the interactions between elements in terms of the interfaces that they support, and the behaviors that they exhibit through these interfaces.

Unity can support multiple, logically separated *application environments*, each providing a distinct application service. Each application environment is represented by an *application manager* element, which is responsible for managing the environment, for obtaining the resources that the environment needs to meet its goals, and for communicating with other elements on matters relevant to the management of the environment. One key responsibility of an application manager is to predict how changes in its allocated resources would affect the environment's ability to meet its goals. We have written application managers for typical web service requests directed to a set of servers by a workload driver or by IBM's WebSphere Edge Server, for applications parallelized through IBM's Topology Aware Grid Services Scheduler, and for our own test applications.

Resource arbiter elements compute allocations of resources to application environments. This is done by obtaining from each application environment an estimate of the value of various possible allocations, and calculating an optimum allocation, as described in Section 5. Resource arbiters also represent the "solution" as a whole (the entire set of application environments, resources, etc.) to the outside world, and are responsible for any overall bootstrapping and maintenance issues.

In the current Unity configuration, the resources being allocated are individual servers. Each server is represented by a *server* element, which is responsible for (among other things) announcing the server's address and capabilities in such a way that possible users of the server can see them.

Each host computer that can support autonomic elements is represented by an *OSContainer* element, which accepts requests from elements to start up certain services or other autonomic elements.

Registry elements, based on the Virtual Organization Registry [5], enable elements to locate other elements with which they need to communicate. This role is analogous to registries in other multi-agent systems (e.g., [2]).

Policy repository elements support interfaces that allow human administrators to enter high-level policies that guide the operation of the system. We describe utility-function based policies below; other policies control simpler aspects of the system's operation, such as reserving a server for testing or freeing it for use.

Sentinel elements support interfaces that allow one element to ask the sentinel to monitor the functioning of another. If the monitored element becomes unresponsive, the sentinel notifies the element that requested the monitoring. Sentinels take part in cluster self-healing, as described in Section 4.

Unity sentinels are designed explicitly for monitoring OGSA services. When a sentinel is asked to monitor a target service, the sentinel periodically reads standard Service Data Elements from that target service to determine if is still functioning. The sentinel then passes the target service status to the requesting service via Service Data.

Unity also has a user interface that allows an administrator to observe and direct the system. The user interface

is a web application consisting of a number of servlets, portlets, and applets, built using IBM's Integrated Solutions Console, an interface framework that is itself built on WebSphere Portal technology. It communicates with the autonomic elements in the system through their defined Web-Services interfaces; it has no privileged access to any component. This allows one to create replacement or alternative user interfaces for Unity without altering any other part of the system.

The Unity user interface allows the user to define high-level policies and utility functions and enter them into the policy repository. It polls the registry and the autonomic elements at regular intervals to obtain current performance values for each application environment, and allows the user to examine the performance of the application environments in the system and the current state of each autonomic element.

The Unity UI is a system-wide management interface, but it is possible to construct user interfaces to specific autonomic elements. One of the goals of Unity is to explore user-interface design patterns in autonomic systems and to study, for instance, the relationship between element-specific user interfaces and broader system interfaces.

3. Goal-driven Self-assembly

One of our design aims for autonomic systems is to self-configure, based on the environment in which they find themselves and the high-level tasks to which they have been set, without any detailed human intervention in the form of configuration files or installation dialogs.

Within Unity, we are experimenting with a technique that we call "goal-driven self-assembly". Ideally, each autonomic element, when it first begins to execute, knows only a high-level description of what it is supposed to be doing (e.g., "make yourself available as an application server", or "join policy repository cluster 17"), and the contact information (Grid Service Handle) of the registry. [1]

The self-assembly process proceeds as follows. When each element initializes, it contacts the registry to locate existing elements that can supply services that it requires. It contacts the elements thus located, and enters into relationships to obtain the needed services. Once the element has entered into all the relationships and obtained all the resources that it needs, it registers itself in the registry, so that it can be located by elements that in turn need its services. This process is not confined to initialization. If an elements need another certain service later on in its lifecycle, it similarly contacts the registry to find available suppliers.

A key service located through the registry is the policy repository, which contains, in principle, everything that an element needs to know beyond the registry address and its own high-level role. As one of its first actions, a newly-initialized element locates and contacts a policy repository, queries it for the policies governing elements acting in its role, and uses the result of the query to make decisions about further configuration and subsequent operation. [2]

Concretely, the first elements to start are the OSContainers and the registry, which are necessary to start the other elements. A bootstrap process then starts the resource arbiter, which decides what other elements need to be started and contacts OSContainers (found in the registry) to arrange for their starting. The policy repository and sentinels register with the registry immediately upon coming up. The arbiter registers with the registry, locates the existing policy repositories and sentinels, and contracts with a sentinel to watch each policy repository. Server elements locate and contact the arbiter to announce themselves, and application environment managers contact the arbiter to have servers allocated to them. None of the elements knows in advance where the others are, or even how many other elements of a given kind will exist.

The above description glosses over some potentially complex issues of bootstrapping. Our current system centralizes much of the bootstrapping process, relying on the resource arbiter to contact OSContainers to bring into being those other elements required by the system. We plan to develop a more dynamic and decentralized bootstrapping protocol in which each element would be responsible for instantiating any other elements that it requires, if none are already available. Another interesting approach would be to retain the resource arbiter function and define a language for solution recipes which would tell the solution manager which elements (or at least which initial elements) to bring up to start the system. It is important that any new approaches properly handle circular dependencies.

A smaller-scale bootstrapping issue is that when the first OSContainer element comes up it cannot register because there is not yet a registry running. In our current design, each OSContainer consults its information about where the registry *should* be, and if that address refers to a registry that the OSContainer could create, it creates it.

Similarly, no element is able to contact a policy repository until both a registry and a policy repository have come up. This means that at least both the OSContainers and the registry must be able to function at least temporarily without a policy repository. In fact all elements should have minimal default policies that suffice at least to get them through the process of waiting for a policy repository to appear, and correctly reporting the error if none ever does.

1 In a commercial-grade version of this technique, each element would also be provided with the security credentials needed to prove its identity to the other elements in the system.

2 In the current Unity implementation, only some of these policies are actually stored in and retrieved from the policy repository. We intend to increase that fraction in the coming year.

4. Self-Healing for Clusters

Another main goal of Unity is to demonstrate and study self-healing clusters of autonomic elements. We have initially implemented this style of self-healing in a single element: the policy repository.

The purpose of a self-healing system is to provide reliability and data integrity in the face of imperfect underlying software and hardware. We approach this by adding function to the policy repository to support joining an existing cluster of synchronized policy repositories, and replicating data changes within that cluster. It is also necessary for the cluster as a whole to detect the failure of one of its elements, and to create a replacement element. Care and consideration must be given to which machine should host this new element. For instance, the current policy in Unity is that two elements in the same cluster should not be hosted on the same machine, and that elements in a cluster should not be instantiated on machines that have previously hosted failed elements in that same cluster.

Unity supports clustering with two operations added to the policy repository element. First, whenever a new or modified piece of policy data is received by one of the policy repositories in the cluster, it is sent to all other repositories. This maintains a consistent view (within a few seconds) amongst all policy repositories. (The algorithm currently used does not have transactional integrity, and race conditions can lead to desynchronization in rare conditions. We intend to address this in the near future.)

Secondly, we have modified the subscription system by which Unity elements learn of changes to their policies. In standard OGSI [13] notification, a single OGSA service (the subscriber) subscribes to a given Service Data Element on a single other OGSA service (the publisher). In Unity, the publisher would be the policy repository. In the event of that policy repository failing, even if its data is still safe and available from the other policy repositories in the cluster, the subscriber is left with no subscription, and is never notified of policy changes. In our modified system, subscriptions themselves (including the identity of the subscriber, the class of data subscribed to, and the member of the cluster currently servicing the subscription) are part of the data replicated between elements of the cluster. When a member of the cluster fails, all of its subscriptions are still recorded in the state data of the surviving cluster members. By reassigning those subscriptions to a surviving member, the system can continue providing notifications to the subscribers.

The self-healing cluster operates as follows. When the Unity system is initialized, the resource arbiter determines how many policy repositories are required (this is nominally done by consulting the system policy, but due to the obvious bootstrapping problem this policy is not stored in the policy repository). The resource arbiter then deploys the required policy repositories, each of which is supplied with its intended role (including the identifier of the cluster that it should join) and the registry address. As each one initializes, it consults the registry to contact the already registered members of the cluster and thereby join the cluster itself, using a simple serial algorithm that avoids most race conditions. The resource arbiter also contracts with the sentinel to monitor these policy repositories.

From this point, whenever one of the policy repositories receives changes to its policy set, the changes are communicated to the rest of the cluster, as discussed above. Similarly, policy repositories within the cluster exchange information about which elements are subscribers to the policy data, and to which policy data they are subscribed.

Now, let us assume that the sentinel determines that one of the policy repositories has failed. Perhaps the software has failed, or perhaps the network connection has been severed. The resource arbiter will be notified by the sentinel of this failure, and will first choose one of the still-functioning policy repositories to take over subscriptions previous handled by the failed one, and notify all cluster members of this reassignment. Then, typically, it will determine that it should replace the failed policy repository. In this case, it will examine the available hosts, and select one upon which to deploy a replacement policy repository. The policy repository is deployed (via a standard interaction with the OS-Container on the target host), and upon initialization, the new policy repository joins the cluster and retrieves a copy of the current cluster state data (policies and subscriptions).

Such clusters are still subject to a significant data replication problem. A more complete solution could be assisted by the use of the failover and data replication features of a database management system. The method is also most effective in the case of simple single-element failures; it is not robust against network partitions or similar problems.

The above description still contains a central point of failure, if the system contains only one sentinel. This problem is easily solved with a cluster of sentinels similar to the cluster of policy repositories. Sentinels within a cluster keep each other's state up-to-date, and provide resiliency against the failure of a single sentinel.

5. Self-Optimization Scenario

In this section, we illustrate how utility functions may be effectively used 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 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.

While utility-based resource allocation is widely studied in MAS and other fields, we find surprisingly that it is little known in real-world computer systems management. Additionally, those approaches proposed in the research literature do not fully meet the needs of autonomic computing. Some approaches [4, 7, 10], require humans to ascribe utility value to low-level resources. However, a truly autonomic system should allow utility to be expressed in terms of the service-level attributes that matter to them or their customers, such as end-to-end response time, latency, throughput, etc. Our approach is to use *service-level utility functions* specifying the business value (e.g., payment/penalty terms of a customer contract) as a function of the service level given to users of the application environment. Some market-based approaches [9, 12, 14] 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. To this end, we encapsulate this special knowledge of application environments inside the application managers, as described below.

5.1. Data Center Scenario

The Unity elements and their relationships bearing directly on self-optimization are illustrated in Figure 1. Each application environment has a service-level utility function, obtained from the policy repository. 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 provided in i and $\mathbf{D}_i$ is the demand in i. Both $\mathbf{S}_i$ and $\mathbf{D}_i$ are vectors that can specify values for multiple user classes. $\mathbf{S}_i$ is particular to i, and can be any viable service metric (e.g., response time, throughput, etc.). Although such service-level specification 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. Control and optimization of a fixed amount of resource within an application environment is handled by a resident application manager. As demand shifts, application manager i can 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 $\mathbf{R}_i$. (Here $\mathbf{R}_i$ represents a vector, each component of which indicates the amount of a specific type of resource.)

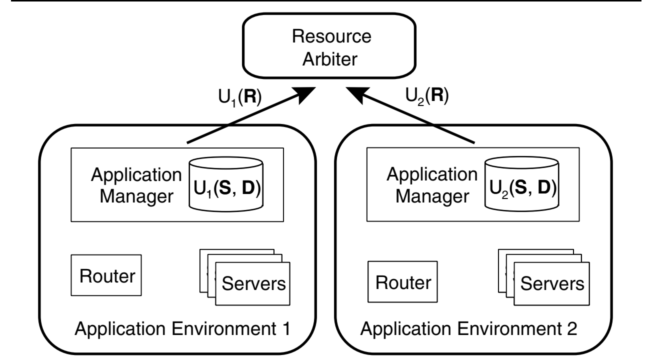

Figure 1. Architecture for self-optimization.

A single resource arbiter allocates resources across different application environments. The arbiter is privy neither to details of how the individual application managers optimize their utility nor 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 resource.[3]

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 in common units. This 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. Our two-level architecture also neatly handles the different time scales appropriate to different types of optimization. Application managers adjust control parameters on a timescale of seconds to respond to changes in demand, while the resource arbiter typically operates on a timescale of minutes, which is more commensurate with switching delays necessitated by flushing out the current workload, changing connections, and installing or uninstalling applications.

5.2. Application Manager Architecture

Figure 2 illustrates the major components and information flows in a single application manager. (As we refer here to only a single manager, we dispense with the

3 If the $\hat{U}(\mathbf{R})$ is too expensive to compute or communicate, it is possible to avoid sending the entire function by having the arbiter query each application manager for a limited set of $\mathbf{R}$ [3].

subscripts.) The application manager receives a continual stream of measured service **S** and demand **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 **C** 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.).

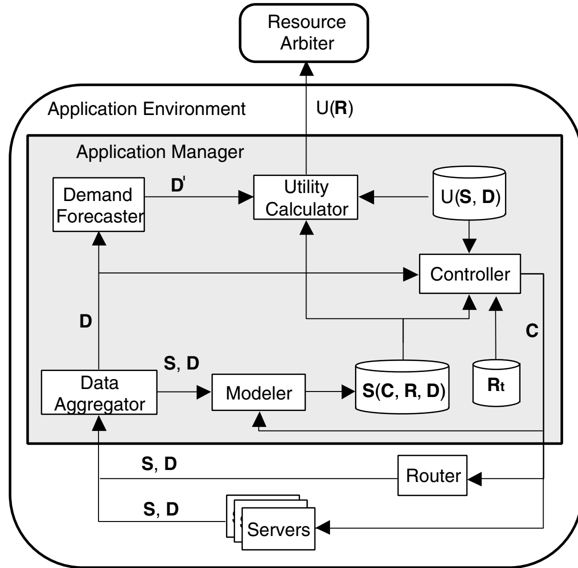

Figure 2. The modules and data flow in an application manager.

The application manager maintains at least three kinds of knowledge: $U(\mathbf{S},\mathbf{D})$, 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 service level obtained by setting control parameters to **C**, given the current **R** and **D**. The model yields a vector of (estimated) service attribute measurements, which could for example represent one or more performance values for each customer class.

The controller optimizes the utility $U(\mathbf{S},\mathbf{D})$ subject to the fixed current $\mathbf{R}_t$. It receives **D** from the data aggregator, and 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:

$$\mathbf{C}^* = \arg\max_{\mathbf{C}} U(S(\mathbf{C},\mathbf{R}_t,\mathbf{D}),\mathbf{D}) \qquad (1)$$

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

The *utility calculator* computes resource-level utility from $U(\mathbf{S},\mathbf{D})$. Since resources are allocated on a relatively long time scale, 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 **D** received from the data aggregator. Methods such as time series analysis and special knowledge of the typical usage patterns of the application may be used. The utility calculator computes

$$\hat{U}(\mathbf{R}) = \max_{\mathbf{C}} U(S(\mathbf{C},\mathbf{R},\mathbf{D}'),\mathbf{D}') \qquad (2)$$

for each possible resource level **R**. To compute the entire $\hat{U}(\mathbf{R})$ essentially requires repeated computation of (1) using each possible hypothetical resource level **R** (rather than the current resource level), and with the predicted demand $\mathbf{D}'$, rather than the current demand **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 on joint observations of $(\mathbf{S},\mathbf{C},\mathbf{R}_t,\mathbf{D})$.

6. Resource Allocation in a Prototype System

To demonstrate the power of our multi-agent system architecture for self-optimization, we show an example of resource allocation based on utility functions in a prototype system. In our example we have two Application Environments with different service-level utility functions based on completely different metrics. The key is that most of the local, 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 to the Resource Arbiter. Here, since **S**, **D** and **R** are all single-valued, we shall replace them with the scalar notation *S*, *D*, and *R*.

Our prototype system runs on a cluster of identical IBM eServer x Series 335 machines running Redhat Enterprise Linux Advanced Server 2.1. The system conforms to Figure 1 using Unity elements as described in Section 2. The two Application Managers, responsible for managing the two Environments, as well as the Arbiter, run on one of the machines; three other machines are available as resources to process workload.

Application Environment A1 handles a transactional workload, running on top of WebSphere and DB2, which provides a realistic simulation of an electronic trading platform. The service-level utility function for A1 is based on the average response time $\mathbf{S}_1$ of the customer requests. More precisely, its service-level utility function $U_1(\mathbf{S}_1,\mathbf{D}_1) = U_1(\mathbf{S}_1)$ is given by a decreasing sigmoid utility function (roughly, a smoothed out step function) with a

maximum value of 1000 for fast response times and a minimum value of zero for slow response times.

Demand for the transactional workload $\mathbf{D}_1$ consists of repeated requests for a web page at a variable rate. To provide for a realistic simulation of stochastic, periodic and bursty web traffic, we use a time-series model developed by Squillante et al. [11] to reset the demand every ~5 seconds.

Given the service-level utility $U_1(\mathbf{S}_1)$, A1 uses a simple system performance model $S_1(C, R_1, D') = S_1(R_1, D_1)$ for each possible number of servers $\mathbf{R}_1$ to estimate the resource-level utility function $U_1(R_1)$. In these experiments, server control parameters C are held constant and the demand forecaster simply returns D. We obtained the performance model by measuring average response time with demand held constant at certain values, and performing linear interpolation between the measured points.

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 and the the utility function $U_2(S_2, D_2) = U_2(R) = \hat{U}_2(R)$ is defined as: $\{U_2(0) = 0, U_2(1) = 200, U_2(2) = 300, U_2(3) = 350\}$.

Results from a typical experiment with the two Application Environments and three servers are shown in Figure 3. 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 $U_1(S_1)$ so that it transitions relatively sharply from 1000 to zero utility, centered at 30ms. The D_1 begins low, allowing A1 to get a low response time and utility of nearly 1000 with one server. At time **a**, D_1 rises, and the manager of A1 changes $\hat{U}(R)$ so that two or more servers are needed to get a high utility. In response, the Arbiter reoptimizes the allocation to give 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. The subsequent decrease in demand allows A1 to obtain low response times with three or fewer servers, and the optimal allocation gives some servers to A1. 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[4].

4 Investigation reveals that they are due to Java garbage collection in WebSphere.

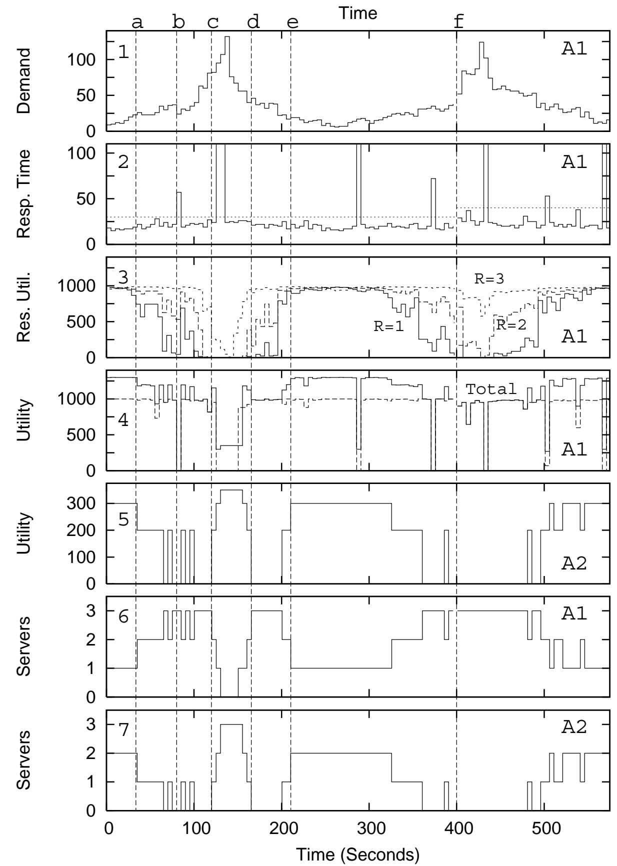

Figure 3. Times series plots of transactional demand rate, attained utilities, and server allocations to each Application Environment, during a sample Unity experiment.

At time **f**, we change $U_1(S)$ in the policy repository so that it transitions more gradually, and is centered at 40ms. A then 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.

7. Conclusions and Future Work

Many researchers in the MAS community have recognized the advantages of an agent-based approach to building deployable solutions in a number of application domains comprising complex, distributed systems. In this paper we put forth self-managing distributed computing systems as a new and promising application domain for MAS ideas. This area is not only natually well-suited to agent-based approaches, but it is also of immense interest and practical importance throughout the entire IT industry.

Our decentralized and distributed approach to autonomic

computing provided the basic principles underlying the development of our Unity software architecture. Through development and experimentation within Unity, we have found it to be a valuable platform for studying and testing ideas about autonomic systems. In particular we found that utility functions provide a powerful and flexible way to allow systems to manage themselves.

We intend to expand Unity to include a wider range of functions and products, and to illuminate more of the large and interesting space of self-managing systems. Also, our prototype workload management system highlights many important practical issues that will need to be addressed in a deployed real-world system. While our implemented system shows feasibility on a small scale, there is much work remaining to be done in scaling the system to handle the complexities encountered in real-world data centers.

In future work, many of Unity's current features will be generalized. For example, we plan to expand the number of supported application environments, and learn what interface extensions may be required. Likewise, self-healing clusters will be expanded to other elements such as registries and arbiters. (This may require innovation to avoid bootstrapping problems.) We are also developing synchronization algorithms for cluster states that have transactional integrity and are robust against race conditions. Additionally, we are enhancing the user interface to include advanced policy and utility function tooling methods, to allow greater exploration within this rich space.

We plan to add flexibility to self-assembly so that the parts can form different useful wholes, closer to the ultimate dynamic vision of self-assembly. This will require standard languages and taxonomies for services offered, dependencies, registry queries, and so on. In complex environments, it will be important to avoid deadlock and circular dependencies during self-configuration. It would also be interesting to develop ways to query the parts as to what the outcome would be of a hypothetical self-assembly.

The use of utility functions will be expanded throughout the Unity system. The self-assembly process, for instance, could use utility functions to decide between various alternate system configurations. System properties like the sizes of self-healing clusters could be derived from higher-level goals (in terms of estimated reliability, say), rather than specified directly. Current hardcoded behaviors, such as bringing up each member of a self-healing cluster on a different host system, could instead be derived from higher-level principles.

Finally, we are also investigating more sophisticated approaches to resource allocation. One particular focus is on developing accurate models of switching costs that may be incurred when reassigning a server, and incorporating such costs in the optimization problem. We are also developing more complex and realistic systems models combining queuing theory, simulation, and novel machine learning methods that allow the models to be learned and refined online, during the operation of the system.

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] D. E. Atkins, W. P. Birmingham, E. H. Durfee, E. J. Glover, T. Mullen, E. A. Rudensteiner, E. Soloway, J. M. Vidal, R. Wallace, and M. P. Wellman. Toward inquiry-based education through interacting software agents. *Computer*, 29(5):69–77, May 1996.

[3] 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.

[4] 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.

[5] 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, 2002.

[6] N. R. Jennings. On agent-based software engineering. *Artificial Intelligence*, 117:277–296, 2000.

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

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

[9] 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.

[10] 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.

[11] 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.

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

[13] 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, 2002.

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