

Utility-based Collaboration among Autonomous Agents for Resource Allocation in Data Centers

Rajarshi Das
IBM Thomas J. Watson
Research Center
Hawthorne, New York 10532
rajarshi@us.ibm.com

Ian Whalley
IBM Thomas J. Watson
Research Center
Hawthorne, New York 10532
inw@us.ibm.com

Jeffrey O. Kephart
IBM Thomas J. Watson
Research Center
Hawthorne, New York 10532
kephart@us.ibm.com

ABSTRACT

Autonomic computing, a proposed solution to the looming complexity crisis in IT, is a realm in which software agents and multi-agent systems can play a critically important role. Conversely, given its importance to a multi-billion dollar industry, it is fair to say that autonomic computing is a *killer app* for agents. Two years ago, we introduced Unity, an agent-based autonomic data center prototype that demonstrated the virtues of agency in autonomic computing applications. We discuss the road to commercialization of Unity, which entails infusing agent concepts into well-established lines of software and middleware, and discuss experiments that establish the commercial viability of utility-based resource allocation. Furthermore we examine the practicality of framing resource allocation in data centers as a collaboration between two agents, each of which is based on a commercially available product.

Categories and Subject Descriptors

H.4 [Information Systems Applications]: Miscellaneous

Keywords

Autonomic computing, resource allocation, data centers, utility functions

1. INTRODUCTION

The complexity of modern-day corporate computing infrastructures makes them very difficult for administrators to install, configure, maintain, and tune. A typical medium- to large-scale enterprise may employ hundreds of applications from half a dozen different vendors running on a combination of mainframes, Linux, and Windows systems. Often, software compatibility issues necessitate running different versions of one vendor's middleware or database management systems within the same IT (Information Technology) environment. The data flows among the applications are often so complex that visualizations of them are too dense

with lines to discern the interdependencies, let alone comprehend them.

IT complexity has several unfortunate consequences, of which we cite just three. First, it significantly raises the cost of making a change once the infrastructure is in place. Even after taking many precautionary measures, administrators frequently make mistakes in adding a new component or in upgrading the middleware on one of the applications with resulting costs that range from thousands to hundreds of thousands of dollars per hour. The high cost and risk of change makes IT systems ponderous—in contradistinction to the increasing call by customers for On Demand computing, which promises flexible and resilient IT systems that support nimble response to new business threats and opportunities. Second, the complexity of individual system components and their interdependencies makes it difficult for administrators to localize, diagnose, and fix problems. Often, the supposed fix does not work, or creates new problems. Third, administrators are finding it ever more difficult to keep pace with the flood of new parameters that get introduced with each new release of products. New books on database management systems from IBM, Oracle, or Microsoft are literally out of date before they are printed. Under such circumstances, it is very difficult to extract the best performance from the system, and system administrators dare not tweak knobs too much for fear of degrading performance or creating other problems.

Most major IT vendors agree that the rapidly growing complexity of IT is a looming crisis that can be averted only if systems become more self-managing. IBM coined the term “autonomic computing” in 2001 to describe its vision of a future in which large-scale IT systems would manage their own behavior in accordance with high-level objectives specified by administrators[4]. Other vendors like Hewlett Packard and Microsoft may use somewhat different terminology, but they agree generally on the importance and urgency of making systems more self-managing.

In short, realizing the vision of autonomic computing is of fundamental importance to the future of the multi-billion dollar IT industry. Moreover, there is every reason to believe that software agent and artificial intelligence technology will play a strong, multi-faceted role in achieving autonomic computing [4]. Inverting the last statement, autonomic computing appears destined to be the long-sought *killer app* for software agents and multi-agent systems [3].

In previous work[10, 13], we discussed and illustrated how an agent-oriented architecture can provide an excellent basis for autonomic computing. Individual self-managing compo-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

AAMAS'06 May 8–12 2006, Hakodate, Hokkaido, Japan.
Copyright 2006 ACM 1-59593-303-4/06/0005 ...\$5.00.

nents can be framed as agents that are responsible for managing their own behavior in accordance with agreements established with other agents. The self-management of the system as a whole (including self-optimization, self-healing, self-configuration, and self-protection) arises from more than just the individual self-managing capabilities of the agents; it is an emergent property of the mutual interactions that occur within the multi-agent system. We demonstrated the advantages of an agent-oriented approach in the context of a simple prototype data center called Unity. Middle agents [9] such as registries and sentinels were used to attain a measure of self-configuration and self-healing. Utility functions expressed in terms of service-level attributes such as response time, used in conjunction with models and optimization techniques, drove efficient allocation of server resources across multiple large-scale applications.

Unity proved to be no exception to the general rule that the flow of ideas from research prototypes to real commercial products is fraught with barriers that are not just technical in nature. One cannot ignore the strong legacy of a successful product line and abandoning it in favor of productizing a multi-agent system designed from scratch. To do so would not just be difficult for the vendor; it would risk alienating an existing customer base that is reluctant to accept radical departures from legacy solutions. The only practical course is to infuse agency gradually into existing products, demonstrating value incrementally at each step along the road towards full agency.

Therefore, during the last year, we have worked with colleagues at IBM Research and with IBM product development teams to implement some of the Unity ideas in two commercially-available products: WebSphere Extended Deployment and Tivoli Intelligent Orchestrator. WebSphere Extended Deployment (henceforth denoted WXD) is an IBM middleware offering that is used to manage routing, CPU and memory allocation, and software module placement in a large-scale multi-tiered application environment given a fixed set of server resources. Tivoli Intelligent Orchestrator comprises a couple of management components, including the Global Resource Manager (henceforth denoted TIO-GRM). TIO-GRM decides how to allocate server resources across multiple application environments, and communicates its decisions to another management component, the Tivoli Provisioning Manager, which provisions, configures, and re-deploys server resources in accordance with the re-allocation decisions made by TIO-GRM.

The remainder of this paper is structured as follows. Section 2 sets the stage for our current work by providing a historical perspective of our past efforts in getting WXD and TIO-GRM to interact with each other. In section 3, we discuss WXD and TIO-GRM and our modifications to them for utility-based collaboration in greater detail. Then, in section 4, we describe two experiments designed to show that, in practical usage scenarios, our scheme allocates server resources in accordance with an administrator's service-level objectives in the face of dynamically changing workloads and objectives. Finally, in section 5, we summarize our main points and provide thoughts on future directions.

2. BACKGROUND

Conceptually, the capabilities of WXD and TIO-GRM are complementary, with WXD managing to service-level objectives as well as possible within fixed resource constraints,

and TIO-GRM operating one level above, deciding how to allocate resources across multiple independent large-scale application environments as outlined in Figure 1.¹

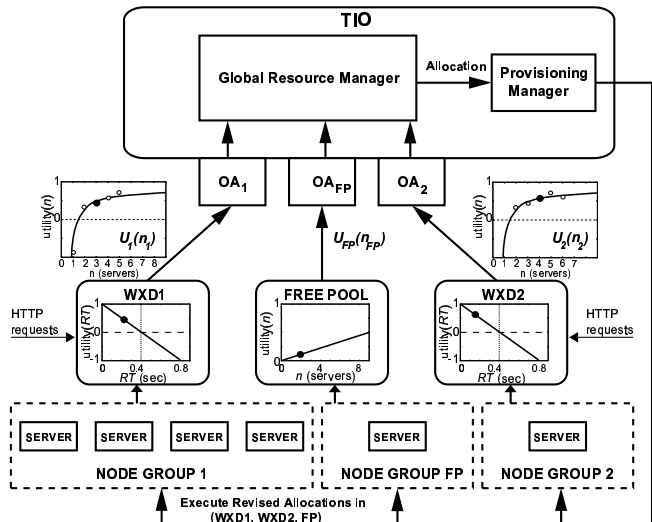


Figure 1: WebSphere Extended Deployment (WXD1 and WXD2), Tivoli Intelligent Orchestrator (TIO), the Objective Analyzers (OAs), and their mutual interactions.

Yet the products as originally conceived did not dovetail very well. This is not surprising, as WXD and TIO-GRM were developed completely independently of one another (indeed, the TIO-GRM technology was obtained by IBM via acquisition) with very different assumptions about environments, span of control, and the nature of the objectives to which they would manage. Therefore there was a great desire on the part of IBM to get these two independently-developed software entities, each capable within their own realm of expertise, to collaborate effectively to achieve system-wide performance optimization.

In recent work [1], we reported early progress in getting WXD and TIO-GRM to interact fruitfully with minimal changes to the existing products. One particularly daunting challenge was the mismatch between the way objectives were expressed by WXD and TIO-GRM. WXD drives its internal behavior using a simple form of service-level utility function expressed in terms of average response time of the various service classes. In other words, an administrator creates a function that describes the value of achieving a given average response time target. This is entirely consistent with the utility-based approach used by Unity. In order to have adopted the Unity approach, WXD should then have computed a resource-level utility $U(n)$ expressing the value of having a given number of server resources at its disposal, and sent this curve to TIO-GRM. TIO-GRM would then have combined similar estimates from other application environments (managed by WXD, or possibly some similar entity) to compute a best overall server allocation that maximized total system utility in each allocation-decision period.

However, the hitch is that TIO-GRM does not view the

¹An application environment might contain a set of applications running on behalf of one customer within a data center, for example.

world in terms of utility. Rather, TIO-GRM expects to receive “Probability of Breach” (PoB) curves, which describe the estimated probability for the application environment to breach its contract as a function of the number of servers with which it is provided. The resource-level utility curves required by Unity and the Probability of Breach curves expected by TIO-GRM are similar in form, but completely incompatible. WXD is similarly incompatible with TIO-GRM—there is no *breach* of contract in WXD; there is only lower or higher utility.

The approach used in [1] was for WXD to output “speed” numbers that hinted at how urgently it needed additional resources. Then, an *Objective Analyzer*—an extension of WXD designed to mediate interactions with TIO—did a highly heuristic transformation of the speed numbers into a probability of breach. The resulting system was shown to work in a qualitative sense, with more resources being supplied to a single application environment when the load was increased. However, it was difficult to quantify the extent to which WXD was receiving the right amount of resource.

In [12], a follow-on to [1], the form of communication between WXD and TIO-GRM was improved. With help from our colleagues at IBM Research, WXD was instrumented to produce from its internal queuing models the information needed to construct the resource-level utility curves $U(n)$. This achievement required WXD’s internal model to operate in a what-if mode, by which it would express the utility it thought it would experience if there were n servers. This is a fairly sophisticated analysis, as it entails some computation of how internal control parameters would be changed if more or less resource were made available. However, a problem remained. Although WXD was now articulating its resource-level utility curve $U(n)$, TIO-GRM was incapable of consuming it: it still expected to receive PoB curves. Accordingly, we developed a heuristic for converting from $U(n)$ to $PoB(n)$ by mapping extrema and midpoints of these curves to one another on the basis of intended behavior. While this scheme was somewhat better at conveying the true intent of the WXD objectives, it was still difficult to quantify the degree to which this was so.

In this paper, we bring our product implementation significant steps closer to the original agent-based vision of the Unity prototype. As we shall describe, we have worked with IBM product developers to create an experimental version of TIO-GRM that accepts utility input directly, eliminating the need for heuristic conversions from utility to PoB. Now TIO-GRM can allocate resources in a manner that truly reflects the intent of the WXD administrator. Thus utility information—and information derived from it via models and optimization—serves as a clean, consistent, universal means for expressing objectives and managing the system to those objectives automatically. Furthermore, we go beyond the original Unity prototype in that we use utility to represent the relative importance of retaining servers in the free pool, a point that will be further explored in section 4.

3. THE COLLABORATING AGENTS

In this section we discuss the two main collaborating agents, based on WXD and TIO-GRM—both of which are commercially available IBM products. Specifically, we use WebSphere Extended Deployment v6.0, which includes the extensions to WXD that supply utility estimates, and we use an experimental version of Tivoli Intelligent Orchestrator

that is based upon TIO v3.1, but extended to support utility functions. As will be discussed, each of these agents is itself composed of multiple autonomous interacting entities that have agent-like properties.

3.1 WebSphere Extended Deployment

WebSphere Extended Deployment (WXD) is an advanced middleware application server environment. While a full enumeration of the capabilities of WXD is beyond the scope of this paper, we will discuss here those features of the environment that are most relevant to the paper.

WXD provides the ability for administrators to set service-level objectives (in the form of performance targets) for the web applications installed within the environment. It provides a load balancing reverse proxy (called the “On Demand Router”, or ODR) that directs requests to individual application servers such that the specified service-level objectives are met. WXD also starts and stops instances of individual application servers as the extant load changes.

A WXD installation consists of a central management server (called the “Deployment Manager”), one or more ODRs, and several backend nodes (physical machines) that run the applications themselves. Administratively, the backend nodes are grouped into one or more “Node Groups”. Each node can be in multiple Node Groups. The user’s applications are installed into “Dynamic Clusters”, which are in turn backed by the Node Groups—each application is installed into one Dynamic Cluster (although each Dynamic Cluster can host many applications), and each Dynamic Cluster is backed by one Node Group.

Each backend node can run multiple instances of WebSphere Application Server. WXD makes decisions at run-time concerning which instances to start and when. The user’s applications are hosted in these individual instances.

The ODR determines the appropriate application for each inbound request by examining the URI. Then, from the set of servers running the application, it selects a server to process the request, taking into account session persistence, etc. The server handles the request in the usual way, and sends the response to the ODR, which relays it to the client.

Clearly, WXD would not be able to meet the service-level objectives if it merely mediated incoming requests. Sometimes, an application is flooded with so many incoming requests that it cannot meet its service-level objectives. Consequently, WXD is also able to start and stop server instances. Thus when the load for a given application is high, WXD can start additional instances of the WebSphere Application server created for the Dynamic Cluster containing the application.

3.1.1 WXD as agents

Rather than having all the management functionality and logic in WXD in a single place, it is distributed through the system as separate components. These components can be regarded as independent, yet inter-connected, agents. The performance management aspects of WXD are handled by three such agents.

The first such agent is the placement control agent—this determines which application server instances should be running at any given time on each node in the system. The approach taken to solving the placement problem is that discussed in [5].

The second such agent is the flow manager—this manages

the various service classes, trading one off against the other to obtain the highest overall system utility. The flow manager takes the current placement (which servers are available for each Dynamic Cluster) as a given, and calculates the concurrency limits for the various Dynamic Clusters that the ODR uses for request assignment. The algorithms used by the flow manager are essentially those described in [6].

The flow manager is particularly relevant to this paper, as it is the component that the TIO-GRM asks for system utilities. The flow manager is not only capable of considering the system as it currently stands, but also of considering possible future allocation changes. The value to WXD of these possible future allocations (in addition to the value of the current allocation) is expressed in terms of a utility function $U(n)$, where n is the node count.

The third such agent adjusts the load balancing weights used in request routing; its function is not pertinent to this paper, and it will not be mentioned further.

The three performance agents are inter-connected, in that information calculated by each agent is used by the other agents. For example, the flow manager agent computes “demand constraints” which the placement control agents use as input to the placement decision.

3.1.2 The WXD Objective Analyzer

As discussed later in Section 3.2, TIO-GRM provides a mechanism by which components called “Objective Analyzers” (OAs) can be plugged in to provide performance information for specialized systems that TIO-GRM is managing.

WXD provides such an OA, in order that higher-level management and administration functionality can be carried out automatically. As TIO-GRM provides a flexible scripting and automation environment for system management, it can be used to carry out higher-level reallocations that WXD cannot do on its own—specifically, TIO-GRM is able to move nodes between Node Groups. In order to do this, as was discussed above, properties of the nodes other than those controlled by WXD need to be changed—for example, database clients might need to be installed or reconfigured so that a node is capable of running the applications that will be deployed upon it once it is a member of a certain Node Group or set of Node Groups.

A hierarchy of management can therefore be seen between WXD and TIO-GRM—WXD handles management within each Node Group, and is aware of low-level performance properties of the applications, whereas TIO-GRM handles management between Node Groups, and between WXD and other systems in the data center.

The OA that currently ships with WXD v6.0 obtains the $U(n)$ function discussed above from WXD for each Node Group (see Section 3.3 for more information on the communications between the various agents), and uses mechanisms discussed in [12] to convert those utility curves to probability of breach curves $PoB(n)$. Some information is lost in the conversion because WXD’s utility values are fit to negative exponential curves with a small number of parameters, which are then mapped very nonlinearly to parameters describing sigmoids that represent the PoB curves required by TIO-GRM. In addition, as is discussed in Section 4, the PoB values are themselves subjected to a nonlinear transformation to “fitness” values after being passed to TIO, which can distort and magnify any inaccuracies introduced by the OA conversion.

In order to support direct use of utility values by TIO, the commercial OA was modified to produce an experimental OA in which the mapping of $U(n)$ to $PoB(n)$ was removed. This removal introduced a slight problem that is not an issue for the commercial OA. TIO-GRM requires input curves $PoB(n)$ or $U(n)$ that cover a broad range of n (from 0 or 1 to some upper bound on the total number of servers). However, WXD does not compute $U(n)$ for all n ; for a variety of reasons it computes $U(n)$ only for a small range of n that does not differ from the current allocation n_t by more than 2 in either direction. In the commercial OA, the partial range is not a problem because $U(n)$ is used as input to a curve-fitting procedure that yields a complete $PoB(n)$ for all n . In the experimental OA, the problem of missing data values $U(n)$ for $n < n_t - 2$ and $n > n_t + 2$ is solved by extrapolating them from the $U(n)$ in the limited range computed by WXD. The resulting utility curve $U(n)$ for *all* n is input to TIO-GRM.

3.2 TIO Global Resource Manager

Tivoli Intelligent Orchestrator (TIO) is an automated resource management solution for corporate and Internet data centers. It allows automatic policy-based management of computing resources by automating three key data center processes: infrastructure provisioning, capacity management, and service level management. TIO allows system administrators to create, customize, assemble and store for further use a large variety of workflows that automate various data center processes from configuring and allocating servers, to installing, configuring and patching software.

The TIO component of interest for this paper is the TIO Global Resource Manager (denoted TIO-GRM), which is charged with making resource allocation decisions that are optimal and sufficiently stable. An optimization component in TIO-GRM performs a tree search over the possible combinations of server allocations and proposes an allocation with the highest overall system “fitness”, taken to be the sum of the individual fitnesses of each application environment that is requesting resources from TIO.

In the commercially available version of TIO-GRM, the fitness $F(n)$ for an individual application environment is a nonlinear transformation of its $PoB(n)$. While TIO-GRM supports any transformation function, the default transform assigns the highest fitness values to PoB values that are in middle range. Typically, this helps prevent underprovisioning and overprovisioning. However, since the fitness is defined strictly within TIO-GRM, there is no way to ensure that the fitness function reflects the intent of the administrator as expressed in the utility functions defined in WXD. Moreover, even if the administrator had access both to WXD and to TIO-GRM, the doubly compounded nonlinear transformations (from utility to PoB in the OA, and from PoB to fitness within TIO-GRM) would make it excessively difficult for an administrator to construct a fitness function that properly reflected his objectives.

Therefore, to allow the utility-based objectives defined in WXD to flow transparently to TIO-GRM, the commercially available TIO-GRM was extended to support utility functions directly. This was achieved by setting $F(n) = PoB(n) = U(n)$. In other words, aside from the extrapolation provided by the OA, the fitness function is no longer an independent, carefully engineered function that is separately defined within TIO-GRM. It is determined com-

pletely by (and essentially equal to) the utility estimates from WXD. The optimization component in the TIO-GRM is quite generic, and therefore required no changes; as always, it simply maximized the sum of the fitness functions of the individual application environments, which in this case is equivalent to maximizing the overall system utility.

3.3 Inter-agent communication

There are two distinct communication paths between WXD and TIO-GRM. The first path conveys utilities estimated by WXD to TIO-GRM. Periodically, when TIO-GRM desires utility estimates, it calls the OA (which is in the same JVM). In turn, the OA makes a JMX call² to WXD requesting from WXD's flow manager the current utility estimates.

The second communication path executes the resource allocation decisions made by TIO-GRM by adding or removing servers in the WXD environments and the free pool. There are two methods, each of which permit TIO to execute OS-level commands on the machines it wishes to modify. The first method involves use of the "Tivoli Common Agent" (TCA), a client-side agent used by various IBM management software. The second method, which was used in our experiments, does not require TCA, but uses SSH (Secure Shell) to execute the commands instead.

When TIO decides to add a node to a Node Group, it does so by first executing commands on the node to add that node to the WXD system ("federating the node"), and then executing commands on the WXD Deployment Manager to move the node into the Node Group. Conversely, when TIO decides to remove a node from a Node Group, it does so by first executing commands on the WXD Deployment Manager to take the node out of the Node Group, and then executing commands on the node to remove that node from the WXD system (this process is known as "unfederating the node").

4. EXPERIMENTS

In this section, we describe two experiments, each of which is based on what we anticipate will be a common usage pattern for the TIO-GRM and WXD agents in deployed systems. In different ways, the two experiments demonstrate efficacious collaboration between these agents to achieve service-level objectives specified by administrators—even in the face of dynamically changing workloads and dynamically changing objectives. First, we describe an experimental setup common to both experiments, and then we discuss each experiment and its results.

4.1 Experimental Setup

The following setup, which applies to both of the experiments, is illustrated in Figure 1. An installation of TIO-GRM was configured to manage two independent instances of WXD, each running its own application with time-varying workload on a single dedicated Node Group contained in a single dedicated Dynamic Cluster. Additionally, TIO-GRM was configured to manage a free pool of homogeneous servers from which the two WXD instances could be supplied. For each WXD instance, the Deployment Manager and the ODR were installed on separate dedicated servers, amounting to four servers in all. (i.e. four servers were used in all for the WXD management).

² [2] describes WebSphere's use of JMX.

We employed the same web-based application for both instances of WXD. Each application consisted of a single service class. For each service class, WXD permits an administrator to establish a simple utility function expressing the value of attaining a given average response time³. The utility function is elicited via a simple template: the administrator inputs a response time target RT_0 and selects one of seven importance levels ranging from very low to very high. These two parameters are mapped to a simple utility function $U(RT)$, in which the utility drops monotonically from 1 at $RT = 0$ to 0 at the response time target RT_0 . For response times greater than RT_0 , the utility is negative and linearly decreasing with a slope governed by the importance level. In all of our experiments, the response target was $RT_0 = 0.4$ sec, and the importance level was selected to be "very high", which translated to a constant slope for all response times. (For lower values of importance, the absolute value of the slope is reduced to reflect a lower penalty for response times above the target.) Thus the utility was given simply by $U(RT) = \frac{RT_0 - RT}{RT_0}$ for all possible values of RT , with $RT_0 = 0.4$ sec.

The demand in each application, quantified by the number of clients sending requests to the ODR, was generated using a single closed-loop load generator [7] with an adjustable number of clients. In principle, each request required the same amount of CPU cycles. The think time for each client in the closed-loop load generator was drawn from an exponential distribution with a mean of 0.125 sec.

A total of six homogeneous server machines were available for our experiments. Each server machine was an IBM eServer xSeries 335 machine with two Intel Xeon 3.06GHz processors (with 512KB of Level 2 cache) configured with hyperthreading enabled. Each machine had 2.5GB of RAM. The properties of the application and the server machines were such that each machine could serve up to 30-40 clients and still meet RT_0 of 0.4 sec.

TIO was installed on a separate machine and the Global Resource Manager was configured to allocate server machines to the two WXD instances based on *resource utility* information $U(n)$ provided by the two OAs every 30 seconds. The free pool was placed on the same footing as the two WXD instances by ascribing a utility u_f to each server in the free pool, i.e. $U_f(n) = nu_f$. In our experiments, we typically set $u_f = 0.05$, although as reported below we also tried setting it to higher ($u_f = 0.4$) and lower ($u_f = 0.01$) values. While it would have been more elegant to represent the third pool by a third (very simple) OA, for the sake of expediency the free-pool utility was computed inside our modified version of TIO-GRM. From discussion with developers and TIO customers, it is clear that there is a good, solid grounding for the use of a free-pool utility. A server that is idling in the free pool may be less costly than one employed by WXD because some hardware contracts specify a lower rate for idle servers, or because multi-server applica-

³WebSphere Extended Deployment supports performance measures other than average response time, such as percentile response time. In these cases, utility functions are defined similarly to what is described here for average response time. We have conducted a similar series of experiments on percentile response time, and we have also experimented with heterogeneous applications and multiple service classes, but for reasons of clarity and space we focus only on the average response time results in this paper.

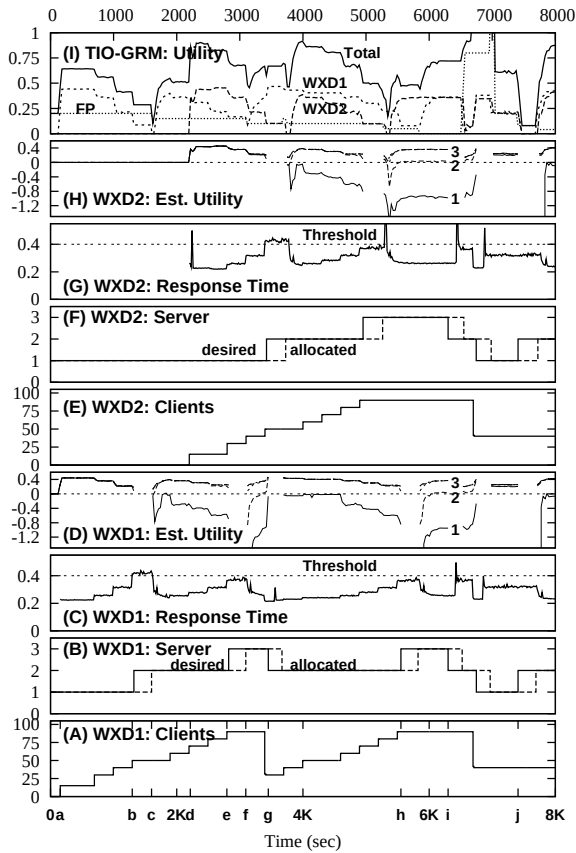


Figure 2: Summary of Experiment 1 in nine time series plots.

tion or middleware software licenses may take into account the number of servers on which the software is installed. An idle machine costs less in terms of power and money to run. Moreover, an idle server is advantageous because, if there is a sudden urgent need, it can be employed more quickly than one that is in currently in service, as there is no need for a potentially lengthy de-installation. Thus in many cases a data center administrator would like to ascribe a utility value to the size of the server free pool.

TIO-GRM guaranteed that at least one server machine would always be made available to each instance of WXD. Thus, based on the current level of load, each WXD was essentially competing with the other WXD instance and the free pool for the remaining four server machines. TIO-GRM's overall goal was to find the allocation that maximized the total system utility, which was taken to be the sum of the utilities derived from the two WXD instances and the free pool.

In all of our experiments, TIO was run in the *automatic* mode, allowing it to execute the resource allocation decisions with no manual intervention.

4.2 Experiment 1

Results from an experiment with TIO-GRM managing two WXD instances, denoted henceforth as WXD1 and WXD2, are summarized in Figure 2. The figure shows nine time series plots over a period of 8000 seconds (2 hours and 20 minutes). From bottom to top, they are:

- (A) the number of clients requesting service in the application installed on WXD1,
- (B) the number of servers scheduled to be allocated and

- the number of servers actually allocated to WXD1,
- (C) the average response time achieved by WXD1,
- (D) the utility values $U(n)$ reported by WXD1 to TIO-GRM,
- (E) the number of clients requesting service in the application installed on WXD2,
- (F) the number of servers scheduled to be allocated and the number of servers actually allocated to WXD2,
- (G) the average response time achieved by WXD2,
- (H) the utility values $U(n)$ reported by WXD2 to TIO-GRM, and
- (I) the total actual system utility obtained by TIO-GRM from WXD1, WXD2, and the free pool (FP).

Notable times are indicated by labeled letters at the bottom of the figure.

At time **a**, each WXD had one allocated server machine as per TIO-GRM's guarantee, and the number of clients for each application was zero. Hence all utility values reported by WXD1 and WXD2 were initially zero. The total system utility obtained by TIO-GRM was derived solely from the free pool, and since there were four machines in the free pool, this value was $0.05 \times 4 = 0.2$.

At time **a**, the number of clients in the application running on WXD1 suddenly jumped to 15. WXD1 already had a server in its Node Group, and after a small transient period the server provided an average response time of about 0.23 sec. The actual utility reported by WXD1 to TIO-GRM was 0.44. Note that WXD1 also estimated that there was virtually no increase in utility $U(n)$ with increasing n .⁴ Because of the utility associated with the free pool, TIO-GRM correctly decided not to allocate any additional servers to WXD1.

From time **a** to **b**, the number of clients in WXD1's application increased in a series of steps. With each such step, it is possible to associate a corresponding increase in response time and a decrease in the utility values. At time **b**, when the number of clients increased to 50, the average response time sharply jumped to over 0.4 sec. This drop in performance also caused a drop in WXD1's utility, and since $U(2)$ was slightly more than $U(1)$, TIO-GRM began the process of allocating an additional server to WXD1. The allocation entails copying application software and other necessary infrastructure information to the new server machine, a process that may take more than 5 minutes.⁵ However, once the server is finally made available to WXD1 at time **c**, there is a drop in response time and a corresponding increase in utility reported by WXD1 to TIO-GRM. The utility of the free pool on the other hand drops to 0.15. Interestingly, even after the server has been allocated to WXD1, there is a substantial transient period before the response time and utilities reached their corresponding near-asymptotic values.

Beyond time **c**, the number of clients associated with WXD1 continues to climb to 90, putting an increasing strain on the two servers in WXD1. In response to further decrease in the reported utility, TIO-GRM decides to allocate the third server to WXD1 at time **e**. A similar series of

⁴This is because the mean queue length was zero for this level of load on a single server. Thus additional servers would not have had any effect.

⁵In its current implementation, the OA stops reporting the estimated utility values to the TIO-GRM while an allocation (or de-allocation) process is in progress. Hence, we see the gaps in the utility values in Figures 2(D) and (H).

events ensues, and once the new server is online at time **f**, the response time drops accordingly.

At time **g**, the trend of increasing clients is reversed, with the client count dropping suddenly to 30. All of the estimated utility values in WXD1 rise dramatically, as a result of which TIO-GRM determines to take away one server from WXD1. In this case, once the server is returned to the free pool, XD1 is still able to provide a response time well below the threshold of 0.4 seconds.

While TIO-GRM and WXD1 were involved in this exchange of server machines, note that back at time **d**, the WXD2 application took on some clients for the first time, and the number of clients began to increase with the passage of time. Accordingly, beyond time **d**, when WXD2's performance begins to sag noticeably, TIO-GRM granted an additional server to WXD2. The allocation of a server to WXD2 and the de-allocation of a server from WXD1 overlap near time **g**, underscoring TIO-GRM's robust ability to handle multiple allocation processes at the same time.

Later, at time **h**, we stress-tested our experimental setup with very high load on WXD1 and WXD2. Soon after TIO-GRM's decision to allocate the last remaining server in the free pool to WXD1 at time **h**, both WXD1 and WXD2 make use of three servers each, enabling them to successfully meet their response time targets and obtain relatively high utility.

To illustrate the ability of the system to respond dynamically to changes in objectives, and to demonstrate the naturalness of the free pool utility, the free pool utility is increased at time **i** from its default value of 0.05 to 0.4. As discussed above, this may represent a change in the licensing structure, or in the perceived advantage of retaining machines in the free pool. Fig. 2 demonstrates that TIO-GRM is able to take this change in stride with no need for stopping and restarting. In this case, TIO-GRM decided to pull one server away from environments WXD1 and WXD2. While this action causes the response times of WXD1 and WXD2 to rise noticeably, and their utilities to fall somewhat, this is more than compensated by the increased utility of the two extra servers in the free pool.

Later, when the load on both WXD1 and WXD2 has dropped, a complementary series of steps is triggered at time **j**, when the system administrator chooses to decrease the value of each individual server in the free pool to 0.01. As a result, TIO-GRM now relinquishes an additional server from the free pool to each WXD.

This experiment demonstrates effective utility-based collaboration among three autonomous agents, resulting in robust, dynamic allocation and de-allocation of server machines that faithfully reflects service-level objectives that may themselves be dynamic in nature.

4.3 Experiment 2

The purpose of the second experiment was to explore the effectiveness and reliability of the collaborative agents in a more dynamic setting with more realistically time-varying workloads. The experimental setup was substantially similar to that of Experiment 1. The main difference was the use of a time-series model of measured web traffic developed by Squillante et al. [8], which provides a realistic emulation of stochastic bursty time-varying demand. The number of clients in the closed-loop load generator was reset every minute according to this model with hard lower and upper thresholds of 5 and 125 clients respectively. The time

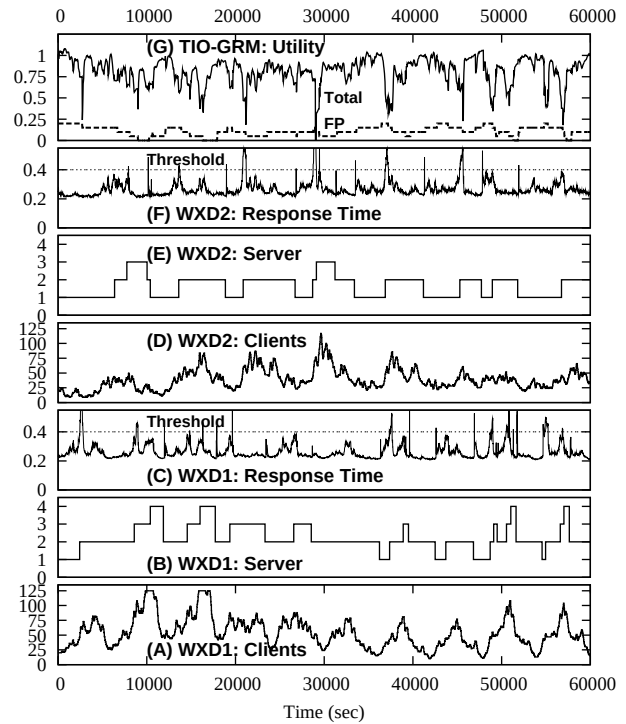


Figure 3: Summary of Experiment 2 in seven time series plots.

series of the workload on each application was generated independently, with the application on WXD1 experiencing slightly higher average load conditions than the application on WXD2.

TIO-GRM was again set in the “automatic” mode, and the experiment ran uninterrupted for a period of over 60000 seconds (16 hours and 40 minutes).

Results from Experiment 2 with TIO-GRM managing WXD1 and WXD2 are summarized in Figure 3. The figure shows seven time series plots; from bottom to top, they are:

- (A) the number of clients requesting service in the application installed on WXD1,
- (B) the number of servers set to be allocated on WXD1,
- (C) the average response time achieved by WXD1,
- (D) the number of clients requesting service in the application installed on WXD2,
- (E) the number of servers set to be allocated on WXD2,
- (F) the average response time achieved by WXD2, and
- (G) the total actual system utility obtained by TIO-GRM from WXD1, WXD2, and the free pool (FP).

The results show that, over the course of over 16 hours, all the agents collaborated successfully. Through a harmonious exchange of servers, each instance of WXD was generally able to meet the response time threshold. Dynamic systems with delayed feedback often suffer from non-linear effects such as hysteresis under time-varying conditions. The XD-TIO-GRM system studied here experiences long transients and allocation delays that can last for over 5 minutes—conditions which under some circumstances can induce thrashing behavior. However, for the time-varying workloads experienced by WXD1 and WXD2 in this experiment, rapid repeated allocation and de-allocation of servers by TIO-GRM was not observed in spite of the allocation delays.

5. CONCLUSIONS

In this paper and in prior work [1, 12], we have argued and demonstrated that software agents and multi-agent systems have a centrally important role to play in autonomic computing, and that, given that it may be the only answer to solving a major problem in the multi-billion dollar IT industry, autonomic computing is a *killer app* for software agents and multi-agent systems. One important proof point in that argument was Unity, an autonomic data center prototype that used an agent-oriented architecture to configure, heal, and optimize itself.

This paper focused on the road to productization for the self-optimizing portion of Unity, which entailed the elicitation, usage, transformation and propagation of utility functions. Direct productization of Unity was not feasible, given the considerable legacy of IBM products that occupy the space of automated provisioning and workload management. Instead, we set about to gradually infuse the most interesting and important elements of Unity into existing products. This paper describes an important breakthrough in that process, in which we have succeeded in implementing in two commercial products the main capabilities that are needed to produce and consume what-if utility information. Two experiments demonstrated effective utility-based collaboration between two products that formerly viewed the world in very different and incompatible terms, resulting in effective cooperative interactions between workload management and provisioning. Indeed, we were even able to take a step beyond the original prototype, showing that management of the free pool (which had not been present in the prototype, and which had been managed heuristically in the product) could be done elegantly, using the same utility-based methods that are used for managing application environments.

Of course, our work is hardly finished. WXD and TIO-GRM still lack features that one would expect in full-fledged agent systems. To cite just one desirable feature, it would be beneficial for the agents to exchange what-if information using more general, flexible semantics. For example, TIO-GRM might find it advantageous to ask for more detailed information on estimated utilities for individual service classes, rather than a single overall utility. We have begun to work on more flexible languages that allow agents to specify and describe hypotheticals. We are also exploring various uses for reinforcement learning techniques in this context [11]. Looking further beyond, we would like to incorporate in these products the other self-managing capabilities in the original Unity that gave rise to its self-healing and self-configuring properties. And looking beyond still further, there is practically infinite scope for employing agent concepts, architectures and technologies—not just for workload management and provisioning, but within autonomic computing systems quite generally.

6. ACKNOWLEDGMENTS

We thank our IBM Research colleagues Giovanni Pacifici, Malgorzata Steinder, Mike Spreitzer, and Asser Tantawi for much discussion and consultation on WebSphere Extended Deployment, without which this project would have been impossible. We are also indebted to U. Gopalakrishnan of IBM India for making several essential enhancements to the workload generator, giving us the flexibility we needed.

7. ADDITIONAL AUTHORS

Additional authors: Hoi Chan (IBM Thomas J. Watson Research Center, email:hychan@us.ibm.com) and Paul Vytas (IBM Software Group, Tivoli, email:pvvtas@ca.ibm.com).

8. REFERENCES

- [1] D. M. Chess, G. Pacifici, M. Spreitzer, M. Steinder, A. Tantawi, and I. Whalley. Experience with collaborating managers: Node group manager and provisioning manager. In *Proceedings of the Second International Conference on Autonomic Computing*, 2005.
- [2] R. Cundiff. System administration for websphere application server v5. http://www-128.ibm.com/developerworks/websphere/techjournal/0302_cundiff/cundiff.html, 2003.
- [3] M. Fisher, J. Muller, M. Schroeder, G. Staniford, and G. Wagner. Methodological foundations for agentbased systems. *Knowledge Engineering Review*, 12(3):323–329, 1997.
- [4] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–52, 2003.
- [5] T. Kimbrel, M. Steinder, M. Sviridenko, and A. Tantawi. Dynamic application placement under service and memory constraints. In *4th International Workshop on Efficient and Experimental Algorithms*, Santorini Island, Greece, May 2005.
- [6] R. Levy, J. Nagarajarao, G. Pacifici, M. Spreitzer, A. Tantawi, and A. Youssef. Performance management for cluster based web services. In *8th IFIP/IEEE International Symposium on Integrated Network Management (IM 2003)*, 2003.
- [7] D. Menasce and V. A. F. Almeida. *Capacity Planning for Web Performance: Metrics, Models, and Methods*. Prentice Hall, 1998.
- [8] 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.
- [9] K. Sycara. Multi-agent infrastructure, agent discovery, and middle agents for web services and interoperation. In *Multi-agents systems and applications*, pages 17–49. Springer-Verlag New York, Inc., New York, NY, USA, 2001.
- [10] G. Tesauro, D. M. Chess, W. E. Walsh, R. Das, A. Segal, I. Whalley, J. O. Kephart, and S. R. White. A multi-agent systems approach to autonomic computing. In *AAMAS*, pages 464–471. IEEE Computer Society, 2004.
- [11] G. Tesauro, R. Das, W. E. Walsh, and J. O. Kephart. Utility-function-driven resource allocation in autonomic systems. In *Second International Conference on Autonomic Computing*, 2005.
- [12] I. Whalley, A. Tantawi, M. Steinder, M. Spreitzer, G. Pacifici, R. Das, and D. M. Chess. Experience with collaborating managers: Node group manager and provisioning manager. To appear.
- [13] S. R. White, J. E. Hanson, I. Whalley, D. M. Chess, and J. O. Kephart. An architectural approach to autonomic computing. In *First International Conference on Autonomic Computing*, 2004.