RC23767 (W0511-012) November 2, 2005
Computer Science

IBM Research Report

Dynamic Collaboration in Autonomic Computing

David M. Chess, Jeffrey O. Kephart, James E. Hanson,
Ian N. Whalley, Steve R. White
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Dynamic collaboration in autonomic computing

Chess, Hanson, Kephart, Whalley, White
IBM Research

CONTENTS

1.1 Introduction

The ultimate goal of autonomic computing is to build computing systems that manage themselves in accordance with high-level objectives specified by humans [15]. The many daunting challenges that must be addressed before this vision is realized fully can be placed into three broad categories[14] pertaining to:

- **Self-managing resources**, the fundamental building block of autonomic systems, which are responsible for managing their own behavior according to their individual policies;
- **Autonomic computing systems**, which are composed of interacting self-managing resources; and
- **Interactions between humans and autonomic computing systems**, through which administrators can express their objectives, monitor the behavior of the system and its elements, and communicate with the system about potential actions that might be taken to improve the system's behavior.

This paper concerns an important aspect of the second of these broad challenge areas: how dynamic collaboration among self-managing resources can give rise to system-level self-management. Research on individual self-managing databases, servers, and storage devices, which significantly predates the coinage of the term *autonomic computing,* has resulted in a steady flow of innovation into products from IBM and other vendors over the years. One can be confident that useful innovations in self-managing resources will continue indefinitely. However, isolated, siloed work on individually self-managing resources, while important, will not by itself lead to self-managing computing *systems*. A system's ability to manage itself relies on more than the sum of the individual self-management capabilities of the elements that compose it. We believe that dynamic collaboration among self-managing resources is a crucial extra ingredient required to support system-level self management.

Why is dynamic collaboration among self-managing resources essential? The primary driver is the ever-growing demand for truly dynamic e-business. For example, IBM's On Demand initiative is a call for more nimble, flexible businesses that can "respond with speed to any customer demand, market opportunity or external threat." [19] Customers want to be able to change their business processes quickly. They want to be able to add or remove server resources without interrupting the operation of the system. They want to handle workload fluctuations efficiently, shifting resource to and from other parts of the system as needs vary. They want to be able to upgrade middleware or applications seamlessly, without stopping and restarting the system. In short, dynamic, responsive, flexible e-business requires for its support a dynamic, responsive, flexible IT infrastructure that can be easily reconfigured and adapted as requirements and environments change.

Unfortunately, today's IT infrastructures are far from this ideal. Present-day IT systems are so brittle that administrators do not dare to add or remove resources, change system configurations, or tune operating parameters unless it is absolutely necessary. When systems are deployed, relatively static connections among resources are established by configuring each of the resources in turn. For example, once it is decided to associate a specific database with a particular storage system, that connection may exist for the lifetime of the application. When changes such as software patches or upgrades are made, it may take days or weeks to plan and rehearse those changes, thinking through all of the possible ramifications to ensure that nothing breaks, and practicing them on a test system devoted to the purpose. The natural result of such justifiable conservatism is a ponderous IT infrastructure, not a nimble one.

How do we make IT infrastructures more nimble? We believe that, beyond making self-managing resources more individually adaptable, we must endow self-managing resources with the ability to dynamically form and dissolve service relationships with one another. The basic idea is to establish a set of behaviors, interfaces, and patterns of interaction that will support such dynamic collaborations, which are an essential ingredient for system-level self-management.

In this paper, we propose a set of requirements, identify a set of standards designed to help support those requirements, and show that systems that partially respect these requirements can indeed exhibit several important aspects of self-management. Section 1.2 sets forth our proposed requirements, which focus on supporting dynamic collaboration among autonomic elements. Next, in section 1.3, we discuss standards and the extent to which they do and do not support those requirements. The next four sections discuss the present status and possible future of two systems that exhibit dynamic collaboration to achieve system-level self-management. The first system, described in section 1.4, is a datacenter prototype called Unity that heals, configures, and optimizes itself in the face of failures, disruptions, and highly varying workload. Unity achieves these self-managing capabilities through a combination of algorithms that support the behavioral requirements, design patterns that introduce new resources such as registries and sentinels that assist other resources, and well-orchestrated interactions. We discuss the benefits that Unity derives from its use of a subset of the requirements and standards. Then, in section 1.5, we speculate on the extra degree of self-management that could be attained were Unity to more fully implement the proposed requirements and standards. The second system, described in section 1.6, focuses more specifically on a particular interaction between two commercially available system components: a workload manager that allocates resources on fine-grained scale (e.g. CPU and memory share) and a resource arbiter that allocates more coarse-grained resources such as entire physical servers. We discuss how the requirements and standards support dynamic collaboration between the workload manager and the resource arbiter, and in section 1.7 we speculate about how their fuller implementation could yield further benefits. We close in section 1.8 with a summary of our recommendations on requirements and standards, and speculations about the future.

1.2 Requirements

The dynamic nature of autonomic systems has profound implications. We do not believe that such dynamism can be achieved with traditional systems management approaches, in which access to resources tends to be idiosyncratic and management tends to be centralized. On the contrary, our approach is to use a **service-oriented architecture** [13], so that there is a uniform means of representing and accessing services, and to make the resources themselves more **self-managing** [24]. Rather than having a centralized database as the primary source of all information about the resources—what kinds of resources they are, what version they are, etc.—we make the resources themselves **self-describing**. Every resource can be asked what kind of resource it is, what version it is, and so on, and it will respond in a standard way that can be understood by other resources. Similarly, resources respond to queries about their capabilities by describing what kinds of services they are capable of offering. We believe that the advantages that come with making resources self-describing will be at least as great as those that have come from self-description mechanisms such as reflection in programming languages [17].

There is, of course, a role for centralized information. Having a database containing descriptions of all available resources is useful, but having it be the primary source of this information is brittle. As soon as a change is made to a resource, the database is no longer correct. We avoid this problem by **making resources responsible for reporting information** about themselves in these centralized databases. When a new resource comes on line, or as it changes in relevant ways, it informs the centralized database of the change. This keeps the database as up to date as possible, without requiring manual updates or periodic polling of all of the resources.

As an essential part of self-management, **resources must govern their actions according to policies**. These policies will typically specify what the resource should do rather than how it should do it. They will specify a performance goal, for instance, rather than a detailed set of configuration paramaters that permit the resource to achieve that goal. It is then left to the self-managing resource to determine how best to attain that goal.

Resources will typically use the services of other resources to do their jobs and, as the system changes over time, which resources are being used will change as well. A database may need a new storage service to handle a new table. A new router may need to be found to take over from a router that failed. **Resources must be capable of finding and using other resources** that can provide the services that they need. **They must be capable of forming persistent usage agreements** with the resources that provide them service. Existing work on service composition, such as [20], will be directly relevant to the challenges here.

There will also be cases where **some resource must determine the needs of other resources**, rather than their capabilities. This is clearly necessary at design time, where both manual design tools and automatic deployment planners must be able to ensure, with a high probability of success, that once the system is in operation all of the necessary resources will be able to obtain the services that they need to function. Similarly, at runtime, a resource's ability to report its needs can provide essential help for system configuration and problem determination.

Owing to the fact that the nature of the services, and the quality of service required, may not be obvious at design time, **resources must be capable of dynamically entering into relationships in which the quality of service is specified** subject to terms that detail constraints on usage, penalties for non-compliance, and so forth. That is, we extend the service-oriented architectural concept of dynamic binding to resources with service level agreements between the resources themselves. The resource providing the service is required to do so consistent with the terms of the service level agreement that it has accepted. The consumer of the service may be similarly required to limit the size, frequency, or other characteristics of requests so that the provider can offer the required service level.

In effect, agreements between resources become the structural glue that holds the system together. They replace the static configuration information of today's system with a mechanism to create similar structures dynamically. Setting up such a structure between two resources involves two steps. In the first step, the resource that wishes to consume a service must find and contact a potential provider. The consumer asks the provider for the details of the agreements that it offers. It then selects one that fits its needs and asks for that specific agreement. If the provider agrees, the agreement is put into place. The second step is for the consumer to use the service of the provider as usual.

It will sometimes be useful for a provider to offer a **default agreement** to any consumer that is allowed to access it. This would be typical, for instance, in a yellow pages service, which provides a "best effort" response to any queries that it gets. If there is a default agreement, it is not necessary to create an agreement as a separate step. Rather, consumers and providers behave as if they had already put an agreement into place. Default agreements do not give the provider the opportunity to say "no," so it is possible for the provider to get more requests than it can satisfy. Explicit, non-default agreements permit the provider to determine which agreements it can satisfy, and put limits on its clients accordingly.

While some agreements will be standalone bilateral arrangements between two resources, others will involve larger collections of resources. In this latter case, when multiple resources need to be involved in an agreement, there is a need for **resource reservations**. These allow the entity setting up the larger whole to reserve all of the necessary resources before actually forming the agreement.

Just as resources can know their own identity and properties, they can know into what agreements they have entered. Just as resources can report their existence and lifecycle state to a centralized resource database, they can report their agreements to a centralized relationship database, and keep this database up to date as new agreements are made and old ones ended.

The dynamic restructuring of the system made possible by agreements gives rise to several interesting problems. As in traditional systems, **security** is important—autonomic systems should limit access to those entities that are authorized to access them. The automated nature of autonomic systems makes this even more important. Consider a lifecycle manager that is responsible for moving a small set of resources through their lifecycle states in preparation for maintenance but, because of a bug, sends a "shut down" command to every resource in the system. Authorization control is an important means of limiting such problems. In addition, because of the dynamic nature of autonomic systems, these authorizations need to be dynamic as well, so that the set of authorized resources can change at run time. To some extent this dynamism of authorization is already present in today's systems, where authorization is typically given to named groups at deployment time, and group membership may change dynamically. However, we expect that the requirements for dynamic authorization in fully autonomic systems will go beyond what can be provided by group membership alone.

In an autonomic systems, system management resources can be asked to manage different resources over time. There are likely to be a variety of discipline-specific managers: performance managers, availability managers, security managers, and so on. These disciplines are not orthogonal; increasing availability by mirroring requires extra resources that could have been used to increase performance. In traditional systems, conflicts between the objectives of the various managers are resolved at design time. A single performance manager is assigned to a group of resources. A decision is made ahead of time to trade off performance for availability. In an autonomic system, however, **conflict avoidance, detection and resolution** must be done at run time, hopefully without requring the intervention of human administrators.

As the system plans for changes, it must be possible to project the impact of those changes. One approach would be to have a centralized model of the entire system and use it to anticipate the results of change. Such a model would have to possess a rich understanding of the resources themselves and their current operating state. Resource models used in this system model would have to be kept synchronized as the resources themselves are updated, or change their lifecycle state or operating characteristics. The approach that we take instead is to require the resources themselves to **respond to queries about the impact of hypothetical changes** that we might make to them. This eliminates the need for a seprate, external model and allows resource designers to handle the queries, and the associated internal processing, in whatever way they want.

1.3 Standards

It is easy to both overestimate and underestimate the important of standards for computing systems. Having a standard for interchanging a particular class of information is important for interoperability, but it provides little or no help with the problem of actually producing or exploiting that information. On the other hand, given any pairwise interaction between computing resources, it is generally easy to design a method for transferring the information required by the interaction between those resources; no standards are required. But *ad hoc* pairwise interaction design does not scale well, and if autonomic computing is to have a real impact, the heterogeneous resources that form autonomic systems must conform to open, public standards. In this section we review the standards most relevant to the requirements discussed in Section 1.2, both to evaluate their suitability and to find what is missing.*

1.3.1 Emerging web service standards

In keeping with our overall service-oriented architecture of autonomic computing systems, all the standards we consider are applicable to systems constructed according to the Web Services Architecture [23]. The fundamental specification for the description of Web Services is the Web Service Description Language (WSDL) [11]. WSDL enables a Web Service to characterize the set of messages it is capable of processing—i.e., to describe its own interfaces. WSDL is a highly flexible XML dialect supporting an extremely wide range of interaction styles, from generic messaging to strongly typed RPC-like invocation. It also has the advantage of being

*Many of the specifications discussed in this section are still under development, and many are faced with competing specifications that provide more or less similar features. While it is impossible to predict the ultimate form the set of standards will take, it is reasonable to assume that it will not fundamentally differ from what is described here.

independent of the particular transport protocol used, thereby permitting designers to select the protocol most appropriate for their needs.

By itself, WSDL does not make a clear distinction between Web Service types and instances. Thus, for example, one common implementation pattern is for a given URL to be associated with a particular service interface as defined by a given WSDL file, but for each message sent to that URL to be handled by a newly instantiated runtime instance of the service. This is insufficient for the needs of long-lived stateful resources. Web Services Addressing (WS-Addressing) [6] fixes this problem by defining a standard form for messaging addresses of *stateful* resources—i.e., the logical "endpoint" address to which messages intended for a given resource instance should be sent.

The long-lived, stateful, active nature of autonomic resources also implies a number of other needs, such as publishing and accessing a resource's state information, publishing and subscribing to events, etc. The Web Services Resource Framework (WSRF) [34, 36, 35, 37, 33] is a constellation of specifications designed specifically to meet these needs in a standardized way. Standardized access to a stateful resource's published state information is given by Web Services Resource Properties (WS-ResourceProperties), which supports both get and set operations for a resource's properties as well as complex XPath-based queries. Web Services Resource Lifetime (WS-ResourceLifetime) provides support for long-lived, but not immortal, resources. The need for registries of resources is met by the Web Services Service Group specification (WS-ServiceGroup).

Sometimes resources need to expose additional information about their interfaces that may not be expressed using the specifications mentioned so far. For example, it may be the case that a certain property of a resource never changes once set, or may be changed by the resource but not by outside entities, or may be settable by others, and so on. Web Services Resource Metadata [25] provides a standard means by which such additional information may be attached to interface operations and properties.

The essential infrastructure for enabling resources to publish and subscribe to asynchronous events is provided by the Web Services Notification (WS-Notification) specificiations. [30, 31, 32] This is particularly important for autonomic computing, in which resources must be able to react quickly to externally-applied changes in their environment, and to report changes in their own state information.

Web Services Security (WS-Security) [26] refers to a collection of related security protocols. For the needs of autonomic systems, we recognize a particular subset of these as being especially useful: resources need to be able to identify the sender of a message and to authenticate that identity; they need to determine the authorization level of the sender, i.e., whether they should pay attention; and they need to be able to communicate without the risk that messages will be intercepted by unauthorized parties. The basic WS-Security standards, defining how data is signed and encrypted and how particular security algorithms are named, are now well established. The higher-level standards, which are required for truly interoperable dynamic security configuration, are still in process.

Built on WSRF, Web Services Distributed Management (WSDM) [28, 29, 27] is another constellation of specifications that meets certain essential needs of autonomic systems. WSDM at present consists of two major parts: MUWS (Management using Web Services) and MOWS (Management of Web Services). At the core of the WSDM specifications is the notion of a *capability*, which is a URI (Uniform Resource Identifier [5]) that identifies a particular, explicitly documented set of operations, resource properties, and functional behavior. Much of WSDM's attention is given to the manageability capabilities of resources—i.e., the ways in which resources may be managed by traditional systems management architectures. These capabilities are clearly useful to managers in an autonomic system. More importantly, the general notion of capabilities, and the WSDM representation of them, is clearly important as a means of identifying what resources are capable of doing.

Arguably the simplest WSDM capability is *identity*, which is something both essential in autonomic systems and conspicuously absent from the the base Web Services and WSRF standards. The notion of resource identity turns out to have subtleties we do not have space to discuss here, but at minimum it should be a unique identifier for a given stateful resource that persists across possible changes in service interface and resource address. Also, since some resources may in fact possess multiple resource addresses, a resource's identity may be used to determine whether or not two different endpoint addresses point to the same resource.

The Web Services Policy Framework (WS-Policy) [3, 7, 4] is a constellation of specifications that partially support autonomic resources' needs for communicating policies. It is primarily an envelope specification, leaving the detailed expression of a policy's content for definition elsewhere.

Web Services Agreement (WS-Agreement) [2] provides the necessary generic support for agreements between autonomic resources, including expression of terms regarding levels of service to be supplied and the conditions under which the agreement applies. It focuses on the outer or envelope format of an agreement document, and on the simplest possible mechanisms for agreement creation and management. It leaves open the detailed content of the agreement's terms, recognizing that it will frequently be resource-specific.

WS-Agreement specifies a format for agreement *templates*, which allow entities to announce a particular set or class of agreements which they are willing to consider entering into. Resources may use this feature to express their default agreements and, more generally, to describe their own collaborative capabilities. It also supports the expression of the ways in which an agreement may be monitored, and the actions that are to be taken when the terms of the agreement are violated. These, taken together with the basic notion of dynamically establishing service commitments at the heart of WS-Agreement, supply important interfaces to help resources avoid, detect, and manage conflicts.

1.3.2 Opportunities

There remain several basic requirements of collaborating autonomic resources that are not obviously met by the Web Service standards reviewed above: describing their specific capabilities, expressing policy content, describing the services they need, supporting resource reservations, and responding to queries about the impact of hypothetical changes. These represent opportunities for further development of standardized interfaces.

The notion of capabilities is both extremely important and highly underdeveloped. The basic approach taken by WSDM, in which a capability is a URI representing a human-readable description of behavior, seems correct as far as it goes. But clearly, if resources are to be able to describe their own capabilities in ways that permit automated matching of service providers with service consumers, something richer than opaque identifiers would be of great value. There are a number of ongoing efforts to design interoperable ways of expressing the semantics of Web Services, including ways of describing hierarchical ontologies of service capabilities [1].

As noted above, WS-Policy leaves the detailed expression of a policy's content largely unrestricted. Autonomic computing requires a semantically rich policy language that will at least support general "if-then" rules and expression of goals and, equally importantly, detailed ontologies for naming the entities to which the polices apply and describing their salient properties.

In order to describe the services that a resource requires (rather than what it provides), a fairly simple specification should be sufficient. In essence, such a description consists of a list of the interfaces and capabilities that a particular resource expects to find in its environment. The former would be references to WSDL documents, and the latter would be capability URIs. It is important to note that this sort of metadata is required both at design time and at run time; if a system is designed without accurate knowledge of what its basic run-time requirements will be, it is small comfort to know that the problem will eventually be detected during operation.

For reservations, WS-Agreement *could* be used, since a reservation is in fact an agreement to provide something at a future time. But WS-Agreement by itself does not descend to the necessary level of detail; additional, reservation-specific content is required, and must therefore also be standardized.

The final item in our list of unmet requirements, estimating the impact of hypothetical changes, is also the least well-developed of them. The usage scenarios specifically require that the resources involved *do not* treat a query as an actual request—i.e., that the recipient does not actually carry out the request but only describes what would happen *if* it were to carry it out—which precludes the use of WS-Agreement. The description of the hypothetical changes may refer to the interface elements that would be used in making the change—e.g., the query can take the form, "What would happen if I were to invoke operation X on you?", where the operation in question is defined in the resource's WSDL—but no such standard exists at present. A standardized representation for the consequences of the change, such as the impact estimates used in the example systems we present in this paper, will also ultimately be required.

1.4 The Unity system

In the next four sections, we will consider two systems that have important autonomic features and at least partially meet the requirements of dynamic collaboration; for each system we will consider both the current state of the system, and ways in which it could be enhanced to meet more of the requirements.

The system described in [9, 21] is a prototype autonomic system called Unity that illustrates some of the features of dynamic collaboration. The resources in the Unity system dynamically describe, discover and configure themselves, form agreements, and exchange impact estimates in order to allocate resources in ways likely to enable the system to meet its goals. Conflicts are avoided either by a centralized utility calculation, or (in a variant not described in the original paper but covered briefly below) by simple cooperative utility estimates.

The IT scenario that the Unity system addresses involves resource allocation between multiple applications. A finite pool of resources must be allocated between two or more applications, where each application provides some service for which there is a time-varying level of external demand. The performance of each application depends on the demand being placed on it, and the amount of resource allocated to it.

Each application is governed by agreements, along the lines described in [16], which specifies the rewards or penalties associated with various possible behaviors of the system. In the Unity system, agreements are expressed in terms ranging from application behavior (for instance, transactions per unit time) to real numbers representing the utility (the "goodness") of a particular level of behavior. The overall success of the system depends on the performance of each application relative to the governing service level agreement.

The management resources of the system must cooperate in order to optimize the overall system performance relative to the set of service level agreements in effect. They do this by discovering resources and forming and maintaining relationships using defined interfaces. To acheive these goals, they make use of other system resources that, while not directly involved in management themselves, provide services and capabilities that the management resources require.

Here we will describe the main components of the system as illustrated in figure 1.1, and briefly describe how it illustrates the principles of dynamic collaboration.

The heart of the Unity system's runtime discovery is the *service registry*. Based in this case on the Virtual Organization Registry defined in [18], its function is analogous to "Yellow Pages" registries in multi-agent systems (see for instance [12]), and very similar to registries based on Ws-ServiceGroup as described above (that specification was not available at the time the system was originally planned).

When each resource in the system comes up, it is provided with little initial configuration information beyond the specific task it is to perform and the address of the service registry. Each resource both registers its own capabilities with the registry,

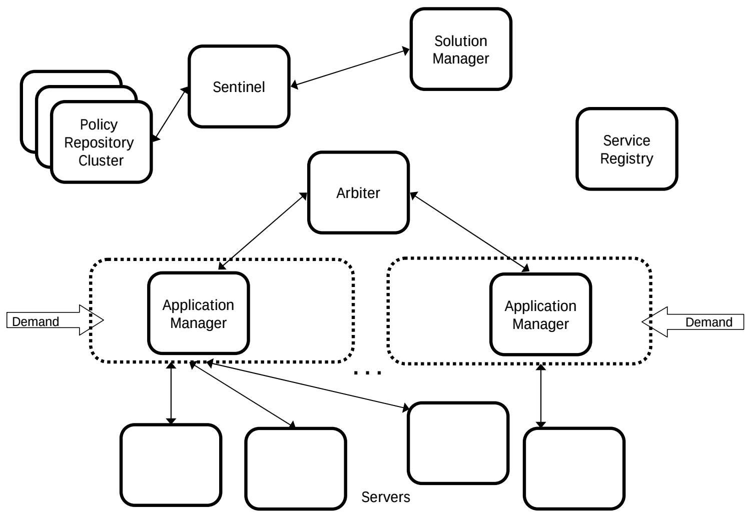

FIGURE 1.1
An abstract schematic of the Unity system. The arbiter component resolves conflicting resource requirements among the application managers, as described in the text. The solution manager uses the sentinel to monitor and heal failures in the cluster of policy repositories. All components access the service registry and policy repository (interaction arrows not shown to avoid clutter), as described in the text.

and queries the registry to find the addresses of the other resourcess from which it will fetch further information (such as policies) and with which it will form agreements to accomplish its goals.

The syntax and semantics of registry information and queries are very simple: each resource creates an entry in the registry containing its address (in the form of a web services endpoint reference), and one or more capability names. Capability names in this system are simple atomic strings, similar to the capability URIs used in the MUWS specifications.

This scheme differs from that used in UDDI tModels [22], in that no hierarchy is represented directly. If one capability is logically a specialization of another, any resource offering the more specialized capability must explicitly register as providing both the more specialized capability and the more general one. This is also the approach taken in [28, 29].

To query the registry, a resource supplies a capability name, and the registry returns a list of addresses of all the resources that have registered as providing that capability. There is an implicit correspondance between capability names and service interfaces: any resource registering itself as having a particular capability is responsible for implementing the corresponding interface. The registry also provides a notification mechanism, by which resources can sign up to be informed when new resources register themselves as providing particular capabilities.

Each application in Unity is represented by an *application manager* resource, which is responsible for the management of the application, for obtaining the resources that the application needs to meet its goals, and for communicating with other resources on matters relevant to the management of the application. In order to obtain the services that it needs, each application manager consults the registry to find resources that have registered as being able to supply those services.

In the system as implemented, the services in question are provided by individual server resources. Each server registers itself as a server with the registry. The management resource that allocates servers to applications is the *arbiter*. It registers itself as able to supply servers, so the application managers can find it. It then consults the registry to determine which servers are available to be allocated. The underlying design scales easily to multiple heterogenous server pools managed by multiple arbiters and distinguished by the capability names that each type of server exposes and each arbiter queries the registry for.

Another important component of the system is the *policy repository*, which holds the service level agreements governing the desired behavior of the system, and other system configuration information. All resources in the system, including the management resources, obtain the address of the policy repository from the registry, and contact it to obtain the policies that apply to them and to subscribe to changes to those policies.

Policies are scoped according to a flat scheme similar to that used to name capabilities in the registry: each resource belongs to a scope that corresponds to the task it is performing, and each policy in the repository belongs to one or more scopes. The policy repository is the primary channel through which human administrators control the system.

The ubiquitous use of the registry and the policy repository enables the system to self-assemble at runtime, without requiring manual configuration but still according to human constraints as expressed in the system policies.

In operation, the arbiter is responsible for managing the resource pool, by controlling which resources are assigned to which application. It does this by obtaining from each application manager an estimate, in terms of utilities as determined by the service level agreements, of the impact of various possible allocations, and calculating an optimum (or expected optimum) allocation of the available resources. It is a key responsibility of each application manager to be able to predict how an increase or decrease in the resources allocated to the application would impact its ability to meet its goals.

The arbiter is not concerned with how the individual application managers make their predictions. It simply uses the results of the predictions to allocate resources. Conversely, the individual application managers do not need to know anything about the activities of the other applications. They need only use their local knowledge of conditions within the application, and their local models of probable future behavior, to make the most accurate predictions they can.

As well as allowing for dynamic reaction to changes at operation time, the comparatively loose and dynamic collaborative management enabled by the service registry, the policy repository, and the direct expression of utilities also has advantages in terms of adding new features and functions. These advantages are illustrated by two additions that were made to the system. In the first (described in more detail in [9]), a *sentinel* resource was added to the system. The sentinel can be asked to monitor other resources for liveness and report when a monitored resource stops responding. A *solution manager* resource was also added, responsible for increasing the reliable operation of the infrastructure as a whole (rather than the utility of the applications); it locates a sentinel via the registry, and enlists it to monitor for failures in any member of a cluster of policy repositries. When notified of a failure by the sentinel, the solution manager starts a new replacement instance of the policy repository to replace the failed one. Adding these features to the system required no changes to the existing resources, which continued to bind and operate as they had before.

The second modification involved giving the application managers the ability to communicate with each other directly about the hypothetical utilities of various server allocations, and to allocate resources properly in the absence of a functioning arbiter. Again the organization of the system as dynamically-bound services allowed us to make this change without fundamentally rearchitecting the existing system.

1.5 Improving the Unity system

The Unity system was designed to explore many of the requirements of dynamic collaboration for autonomic computing, but as implemented it does not include all of the requirements listed in section 1.2. It does not implement significant inter-component security, for instance, or default agreements, both of which would have obvious advantages for the system.

Perhaps the most significant feature that it does not explore is rich service level agreements between resources in the system. The most expressive service level agreements in Unity are logically outward-facing, concerned with the rewards and penalties that accrue to the system as a whole with the behavior of the applications. The agreements between the Unity resources are represented as simple, named atomic relationships, without the internal structure of a service level agreement. Increasing the richness of the inter-resource agreements by adding structured and parameterized agreement terms and utilities (rewards and penalties), based on explicit quality of service measures, would allow the resources of the system to make better-informed decisions about both their internal operations and their dynamically changing relationships.

Similarly, the polices and agreements governing the behavior of the solution manager are extremely simple; the human administrator must specify relatively low-level details such as the number of replicated copies of the policy repository that should be present in the system. As described in [10], we have designed modifications to the system that would allow the administrator to specify policies in higher-level terms (such as effective availability), and automatically derive the more detailed policies from those. The general problem of deriving detailed IT polices from higher-level business policies is one of the major challenges of autonomic computing.

1.6 Node group manager and provisioning manager

The second example we will consider, illustrated in Fig. 1.2, is described more fully in [8]. It consists mainly of two management resources, each with some autonomic characteristics, which collaborate to achieve a level of overall system self-management. While this system as implemented contains fewer dynamic collaboration features than the Unity system, it has the advantage of being based on commercially available management software used in real data centers rather than on research prototypes. After describing the system as it exists, we will consider how the emerging manageability standards could be used to increase the level of dynamism it exhibits.

The first manager in this system, a node group manager implemented in a middleware application server, uses modeling and optimization algorithms to allocate server processes and individual requests among a set of server machines ("nodes") grouped into node groups. It also estimates its ability to fulfill its service-level objectives as a function of the number of nodes potentially available to each node group. [†]

[†]The node group manager in question is a version of WebSphere Extended Deployment, and the provi-

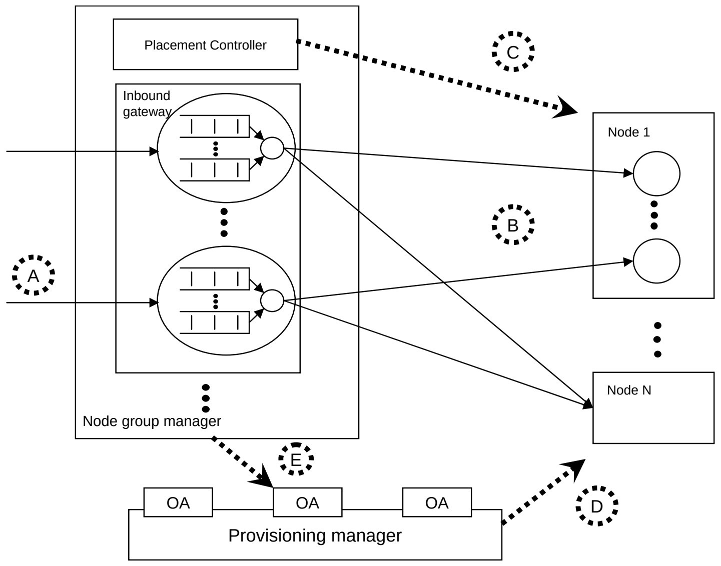

FIGURE 1.2
Overall system structure (described in detail in the text). Incoming requests (A) are queued by the gateway, and dispatched (B) to application servers when resources are available to service them. On a longer timescale, the Placement Controller (C) determines which application servers should run in which nodes. In the longest timescale, the provisioning manager determines (D) which nodes to assign to which node groups by consulting its objective analyzers, which receive estimates (E) from the node group manager (for simplicity only one node group is shown here).

All of the nodes in a given node group share a set of properties, such as installed operating system, network connectivity, support libraries, and so on. The node group manager is responsible for directing and balancing the traffic within each node group, and for allocating server processes to the nodes within the group, but it is not able to carry out the provisioning actions necessary to move nodes from one node group to another. Additionally, the node group manager has only a local view of the set of node groups for which it is responsible. It cannot make higher-level decisions about the allocation of nodes between the demands of its own node groups and those of other processes in the larger data center.

The second manager in the system, a provisioning manager, implements another layer of management above the node group manager. It has the knowledge necessary to move nodes from one node group to another through potentially time-consuming provisioning actions, and it can balance the competing demands of multiple managers operating at the level of the node group manager. These may be instances of the node group manager, or of other managers that can provide the required performance estimates. On the other hand, the provisioning manager does not have the node group manager's real-time knowledge of the traffic within each node group.

The two management layers are thus complementary. The two managers perform a very basic form of "collaboration", in that the estimates produced by the node group manager allow the provisioning manager to more effectively allocate the resources that it provisions, and the actions of the provisioning manager give the node group manager more servers to work with.

The provisioning manager utilizes plugins called "objective analyzers" (OA) to determine the utilization of the various systems it is managing. In operation, the provisioning manager periodically queries the node group managers and each of the other managers at the same level, using the objective analyzers to convert the information from those managers into a form that the provisioning manager understands and can compare across the managers. Specifically, from each of those managers, the provisioning manager requests data estimating, for each potential level of resources that might be allocated to the manager's application, the probability that that the application's service level agreement will be breached if the application is given that level of resources.

This is analogous to the information provided by the application managers to the arbiter in the Unity system described above, except that it is expressed in terms of probabilities of service level agreement breach rather than in terms of resulting utilities. This is a small but interesting difference. In Unity, the application managers, rather than the arbiter, are aware of the actual agreements in effect. Since each application manager is aware in more detail of the behavior of its application, service level agreements evaluated by application managers can potentially reflect knowledge of detailed application behavior that is not normally accessible to a central arbiter. On the other hand, if the central arbiter is aware of individual service level agreements, it can potentially make richer tradeoffs among them. We are currently experimenting

sioning manager is a version of IBM Tivoli Intelligent Orchestrator.

with variants of this system to explore the practical effects of where this information is maintained.

The system described in this section displays dynamic self-management at operation time. In the next section, we will outline how other aspects of the system, such as discovery and binding, could be made similarly dynamic.

1.7 Improving the node group and provisioning managers

In light of the discussion in Section 1.3, it is instructive to consider an idealized version of the node group and provisioning manager discussed in Section 1.6. How might a future version of that system make fuller use of the architecture of dynamic collaboration and the emerging Web Services standards, and what advantages would that future version offer over today's?

1.7.1 Discovery

One of the most obvious advantages of widespread adoption of the WS-Addressing and MUWS standards will be the enablement of dynamic binding. In the current system, the provisioning manager must be manually configured with information about the node group manager. It must be told where the node group manager is, what the nodes in the system are, and what the initial node allocation is. Dynamic binding and discovery would enable the provisioning manager to simply detect these properties of the system, and configure itself appropriately.

Therefore, the provisioning manager would be notified of the arrival of the node group manager by a service registry. The registry would be WS-ServiceGroup based, and would use WS-Notification to alert interested parties about changes to the registry membership. This would enable the system to configure itself automatically in terms of the binding between the managers, eliminating an unnecessary manual step. It would also allow other system resources, not designed to work with these specific products, to take part in the system, either as a replacement for one of the managers, or as third parties taking advantage of the information in other ways. One simple example of this is a system visualization tool that would present the content of the registry to a human administrator.

1.7.2 Negotiation and binding

When the provisioning manager receives notification from the registry that a new node group manager has entered the system, the provisioning manager would then contact the node group manager to determine whether or not to enter into a management agreement with it. The node group manager would expose agreement templates conforming to the WS-Agreement specification that describe the kinds of agreements into which it is willing and able to enter.

WS-Agreement also provides a simple way for the provisioning manager to propose an agreement, and for the node group manager to respond. How the two managers determine internally whether or not to collaborate is of course outside the scope of WS-Agreement. The content of the agreement indicates that the node group manager will provide the provisioning manager with breach probability estimates, and the provisioning manager will in turn control the set of servers with which the node group manager works. WS-Agreement does not specify a content language in which to express this. Further standardization work is required to establish the detailed discipline-specific conventions required here.

There are several reasons why the provisioning manager and the node group manager might decide not to enter into a management agreement. The node group manager might already have entered such an agreement with an alternate provisioning manager. Or, the provisioning manager might not be able to manage the type of nodes that the node group manager needs. For the purposes of our scenario, however, we assume that they do enter into a management agreement.

1.7.3 Data gathering

At this point, the provisioning manager and the new node group manager are bound together. They have a management agreement in place that permits the provisioning manager to make allocation changes to the nodes used by the node group manager. Now the two managers must proceed to actually manage the overall system.

In the current implementation, the provisioning manager polls the node group manager periodically, requesting information concerning the current performance of the node group manager's overall system. In the future system, the node group manager could make these performance statistics available via WS-ResourceProperties, and the provisioning manager would subscribe via WS-Notification to receive updates when these statistics change.

When the provisioning manager decides to change the allocation of the nodes under the control of the node group manager, it would then tell the node group manager to start or stop using a set of nodes. During the formation of the initial agreement, the provisioning manager would have examined the node group manager's manageability characteristics, as specified by the MUWS standard, to ensure that it has the interfaces corresponding to those operations. Here again, the MUWS standard tells us how to expose and access information about manageability characteristics, but it does not give us a specific URI corresponding to the interface that we need. Futher standardization work is required at the discipline level to establish a convention here.

The case where the provisioning manager decides to remove nodes is illuminating. In this case, it would be helpful for the provisioning manager and the node group manager to collaborate on the decision as to which nodes to remove. In the current implementation, for instance, it is often the case that certain nodes under the control of the node group manager require less effort to remove than other nodes. A more detailed exchange of information about the expected impacts of various possible changes would allow the system as a whole to optimize itself even more effectively.

Again, any of the interactions described here could be implemented via *ad hoc* and non-standard languages and protocols. But by using open interoperability standards such as the Web Services standards, resources from different vendors can be used easily, programs that were not specifically designed to work together can collaborate, and new functions not originally anticipated can be composed from existing building blocks.

1.8 Conclusions

While neither system described in this paper is fully self-managing, both support the central thesis: dynamic collaboration is an essential ingredient of system self-management. In the Unity datacenter prototype, system components register themselves (along with a very simple description of their capabilities) to a registry, enabling components to find the services they need. A very simple form of negotiation ensues, resulting in an agreement that forms the basis for a relationship that persists until it is no longer required. Even the relatively rudimentary mechanisms for discovery, resource description, negotiation and agreement make it possible for Unity to assemble itself and exhibit a type of self-healing. Experience with the second system shows that, by adding a thin layer of collaborative capability to two commercially available components that were not originally designed to work together, the resulting system can do an effective job of coordinating optimization of system resources at two levels of granularity.

Thus an encouraging lesson can be drawn from experimental observations of the two systems: a modest degree of system-level self-management can be achieved without fully observing all of the requirements listed in section 1.2. This suggests that, although many challenges remain, good progress towards the ultimate vision of autonomic computing can be made long before all of those challenges are met.

Moreover, this paper's analysis of the strengths and shortcomings of the two experimental systems and the existing body of standards suggests several useful avenues for further work in standards and technology that would bring about a greater degree of system self-management. Many of these center around improving the richness of the dynamic interactions among self-managing resources. Both systems employed very simplistic methods for describing resource needs and capabilities, consisting of a flat mapping between URIs and human-readable descriptions of capabilities and requirements. This very simple language supported some degree of flexibility, such as the ability to add new types of system components without changing the existing ones. Yet it seems likely that richer semantics, accompanied by correspondingly more sophisticated reasoning algorithms, would enable better, more flexible matching between needs and capabilities, and would also form a basis for much more sophisticated forms of negotiation and agreement among resources. Indeed, the negotiation and agreement employed in both systems was quite rudimentary. As suggested in section 1.5, increasing the richness of the description of agreements from a simple named atomic relationship to a full-fledged WS-Agreement document would be a tremendous step forward. However, even this would not provide the ultimate solution, as the standards and technologies required to support negotiation of such agreements do not yet exist.

As a final observation, the standards mentioned here, and related standards too numerous to list, are helpful in improving interoperability and in supporting a degree of dynamic collaboration. Yet, quite generally, they fall short of what is required in the long term because many of them are essentially envelope or data-container standards. In other words, given a description of an interface, a capability, a possible agreement, or an event description, they tell one how to communicate that description in a self-defining way. Much more work is needed to pin down exactly how to represent capabilities—for example, the fact that a given node group manager is able to produce a particular sort of breach probability estimate, or that a particular router is able to handle a specific packet throughput—in ways that fit within these envelopes. Unifying existing discipline and resource-specific standards with the relevant Web Services standards, and devising new ones where they are needed, will be among the most important drivers of further progress in autonomic computing.

1.9 References

[1] R. Akkiraju, J. Farrell, J. Miller, M. Nagarajan, M. Schmidt, A. Sheth, and K. Verma. Web service semantics — WSDL-S. Technical report, A joint UGA-IBM Technical Note, April 2005.

[2] Alain Andrieux, Karl Czajkowski, Asit Dan, Kate Keahey, Heiko Ludwig, Jim Pruyne, John Rofrano, Steve Tuecke, and Ming Xu. Web Services Agreement Specification (WS-Agreement). http://www.ggf.org/Meetings/GGF12/Documents/WS-AgreementSpecification.pdf, 2004.

[3] Siddharth Bajaj, Don Box, Dave Chappell, Francisco Curbera, Glen Daniels, Phillip Hallam-Baker, Maryann Hondo, Chris Kaler, Dave Langworthy, Ashok Malhotra, Anthony Nadalin, Nataraj Nagaratnam, Mark Nottingham, Hemma Prafullchandra, Claus von Riegen, Jeffrey Schlimmer, Chris Sharp, and John Shewchuk. Web services policy framework (WS-Policy). ftp://www6.software.ibm.com/software/developer/library/ws-policy.pdf, 2004.

[4] Siddharth Bajaj, Don Box, Dave Chappell, Francisco Curbera, Glen Daniels, Phillip Hallam-Baker, Maryann Hondo, Chris Kaler, Ashok Malhotra, Hiroshi Maruyama, Anthony Nadalin, Mark Nottingham, David Orchard, Hemma Prafullchandra, Claus von Riegen, Jeffrey Schlimmer, Chris Sharp, and

John Shewchuk. Web services policy attachment (WS-PolicyAttachments). ftp://www6.software.ibm.com/software/developer/library/ws-polat.pdf, 2004.

[5] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifier (URI): Generic syntax. http://www.ietf.org/rfc/rfc3986.txt, 2005.

[6] Don Box, Erik Christensen, Francisco Curbera, Donald Ferguson, Jeffrey Frey, Marc Hadley, Chris Kaler, David Langworthy, Frank Leymann, Brad Lovering, Steve Lucco, Steve Millet, Nirmal Mukhi, Mark Nottingham, David Orchard, John Shewchuk, Eugne Sindambiwe, Tony Storey, Sanjiva Weerawarana, and Steve Winkler. Web Services Addressing (WS-Addressing). http://www.w3.org/Submission/ws-addressing, 2004.

[7] Don Box, Maryann Hondo, Chris Kaler, Hiroshi Maruyama, Anthony Nadalin, Nataraj Nagaratnam, Paul Patrick Claus von Riegen, and John Shewchuk. Web services policy assertions language (WS-PolicyAssertions). ftp://www6.software.ibm.com/software/developer/library/ws-polas.pdf, 2002.

[8] D. 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.

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

[10] David M. Chess, Vibhore Kumar, Alla Segal, and Ian Whalley. Work in progress: Availability-aware self-configuration in autonomic systems. In Akhil Sahai and Felix Wu, editors, *DSOM*, volume 3278 of *Lecture Notes in Computer Science*, pages 257–258. Springer, 2004.

[11] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl, 2001.

[12] E.H. Durfee, D.L. Kiskis, and W.P. Birmingham. The agent architecture of the university of michigan digital library. In *IEEE/British Computer Society Proceedings on Software Engineering (Special Issue on Intelligent Agents)*, February 1997.

[13] H. He. What is service-oriented architecture? http://webservices.xml.com/pub/a/ws/2003/09/30/soa.html, 2003.

[14] Jeffrey O. Kephart. Research challenges of autonomic computing. In *Proceedings of the 27th International Conference on Software Engineering*, pages 15–22, 2005.

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

[16] A. Leff, J.T. Rayfield, and D. Dias. Meeting service level agreements in a commercial grid. *IEEE Internet Computing*, July/August 2003.

[17] Pattie Maes. Concepts and experiments in computational reflection. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 147–155, New York, NY, USA, 1987. ACM Press.

[18] J. Nick, I. Foster, C. Kesselman, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. Technical report, Open Grid Services Infrastructure WG, Global Grid Forum, June 2002.

[19] Stephen Shankland.

[20] B. Srivastava and J. Koehler. Web service composition — current solutions and open problems. In *ICAPS 2003*, 2003.

[21] Gerald Tesauro, David M. Chess, William E. Walsh, Rajarshi Das, Alla Segal, Ian Whalley, Jeffrey O. Kephart, and Steve R. White. A multi-agent systems approach to autonomic computing. In *AAMAS*, pages 464–471. IEEE Computer Society, 2004.

[22] Introduction to UDDI: Important features and functional concepts. http://uddi.org/pubs/uddi-tech-wp.pdf, 2004.

[23] W3C Web Services Architecture Working Group. Web services architecture. http://www.w3.org/TR/ws-arch/, 2004.

[24] Steve R. White, James E. Hanson, Ian Whalley, David M. Chess, and Jeffrey O. Kephart. An architectural approach to autonomic computing. In *First International Conference on Autonomic Computing*, 2004.

[25] Web Services Resource Metadata 1.0 (WS-ResourceMetadataDescriptor). http://www.oasis-open.org/committees/download.php/9758/wsrf-WS-ResourceMetadataDescriptor-1.0-draft-01.PDF, 2004.

[26] Web Services Security: SOAP Message Security 1.0 (WS-Security 2004). http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf, 2004.

[27] Web Services Distributed Management: Management of Web Services (WSDM-MOWS) 1.0. http://docs.oasis-open.org/wsdm/2004/12/wsdm-mows-1.0.pdf, 2005.

[28] Web Services Distributed Management: Management Using Web Services (MUWS 1.0) Part 1. http://docs.oasis-open.org/wsdm/2004/12/wsdm-muws-part1-1.0.pdf, 2005.

[29] Web Services Distributed Management: Management Using Web Services (MUWS 1.0) Part 2. http://docs.oasis-open.org/wsdm/2004/12/wsdm-muws-part2-1.0.pdf, 2005.

[30] Web Services Base Notification 1.3 (WS-BaseNotification). http://www.oasis-open.org/committees/download.php/13488/wsn-ws-base_notification-1.3-spec-pr-01.pdf, 2005.

[31] Web Services Brokered Notification 1.3 (WS-BrokeredNotification). http://www.oasis-open.org/committees/download.php/13485/wsn-ws-brokered_notification-1.3-spec-pr-01.pdf, 2005.

[32] Web Services Topics 1.2 (WS-Topics). http://docs.oasis-open.org/wsn/2004/06/wsn-WS-Topics-1.2-draft-01.pdf, 2004.

[33] Web Services Base Faults 1.2 WS-BaseFaults). http://docs.oasis-open.org/wsrf/wsrf-ws_base_faults-1.2-spec-pr-02.pdf, 2005.

[34] Web Services Resource 1.2 (WS-Resource). http://docs.oasis-open.org/wsrf/wsrf-ws_resource-1.2-spec-pr-02.pdf, 2005.

[35] Web Services Resource Lifetime 1.2 (WS-ResourceLifetime). http://docs.oasis-open.org/wsrf/wsrf-ws_resource_lifetime-1.2-spec-pr-02.pdf, 2005.

[36] Web Services Resource Properties 1.2 (WS-ResourceProperties). http://docs.oasis-open.org/wsrf/wsrf-ws_resource_properties-1.2-spec-pr-02.pdf, 2005.

[37] Web Services Service Group 1.2 (WS-ServiceGroup). http://docs.oasis-open.org/wsrf/wsrf-ws_service_group-1.2-spec-pr-02.pdf, 2005.