AN AUTONOMIC WEB SERVICES ENVIRONMENT USING A REFLECTIVE AND DATABASE-ORIENTED APPROACH

F. Zulkernine, W. Powley, W. Tian, P. Martin, T. Xu and J. Zebedee
School of Computing, Queen's University, Kingston, ON K7L 3N6, Canada
{farhana, wendy, tian, *martin, ziqiang, zebedee}@cs.queensu.ca*

ABSTRACT

The growing complexity of Web service platforms and their dynamically varying workloads make manually managing their performance a tough and time consuming task. Autonomic computing systems that are self-configuring and self-managing have emerged as a promising approach to dealing with this increasing complexity. In this paper we propose a framework for an Autonomic Web Services Environment (AWSE) constructed using a collaborative framework of Autonomic Managers that leverage each component in the AWSE with self-managing capabilities. We present a unique approach to designing the Autonomic Managers for the AWSE framework by combining reflective programming techniques with extended database functionality. The validity of our approach is illustrated by a number of experiments performed using prototype implementations of our autonomic managers. The experimental results demonstrate self-tuning capabilities of the autonomic managers in achieving preset performance goals geared towards satisfying a predefined Service Level Agreement for an Autonomic Web Services environment.

Keywords: Autonomic, Reflective, DBMS, Web Service, Management, AWSE

1   INTRODUCTION

Web services are autonomous software that are remotely accessible by standard interfaces and communication protocols and provide specific services to the consumers. Due to the potential of building complex software systems crossing organizational boundaries by orchestrating multiple Web services, they are now well accepted in Enterprise Application Integration (EAI) and Business to Business Integration (B2Bi) [8]. Performance plays a crucial role in promoting the acceptance and widespread usage of Web services. Poor performance (e.g. long response time) implicates loss of customers and revenue [1]. In the presence of a Service Level Agreement (SLA), failing to meet performance objectives could result in serious financial penalties for the service providers. As a result, Quality of Service (QoS) is of utmost importance, and has recently gained a considerable amount of attention [4], [25].

Accessibility and functionality information of Web services are described in WSDL (Web Service Description Language) [32] documents, which are published or discovered via a UDDI (Universal Description, Discovery and Integration) [29] registry. SOAP (Simple Object Access Protocol) [24] is the most common message passing protocol used to communicate with Web services.

A Web service hosting site typically consists of many individual components such as HTTP servers, application servers, Web service applications, and supporting software such as database management systems. If any component is not properly configured or tuned, the overall performance of the Web service suffers. For example, if the application server is not configured with enough working threads, the system can perform poorly when the workload surges. Typically components such as HTTP servers, application servers or database servers are manually configured, and manually tuned. To dynamically adjust resources in an ever-changing environment, these tasks must be automated.

Unacceptable Web service performance results from both networking and server-side issues. Most often the cause is congested applications and data servers at the service provider's site as these servers are poorly configured and tuned. Expert administrators, knowledgeable in areas such as workload identification, system modeling, capacity

planning, and system tuning, are required to ensure high performance in a Web service environment. However, these administrators face increasingly more difficult challenges brought by the growing functionalities and complexities of Web service systems, which stems from several sources:

- *Increased emphasis on Quality of Services*
  Web services are beginning to provide Quality of Service features. They must guarantee their service level in order that the overall business process goals can be successfully achieved.
- *Advances in functionality, connectivity, availability and heterogeneity*
  Advanced functions such as logging, security, compression, caching, and so on are an integral part of Web service systems. Efficient management and use of these functionalities require a high level of expertise. Additionally, Web services are incorporating many existing heterogeneous applications such as JavaBeans, database systems, CORBA-based applications, or Message Queuing software, which further complicate performance tuning.
- *Workload diversity and variability*
  Dynamic business environments that incorporate Web services bring a broad diversity of workloads in terms of type and intensity. Web service systems must be capable of handling the varying workloads.
- *Multi-tier architecture*
  A typical Web service architecture is multi-tiered. Each tier is a sub-system, which requires different tuning expertise. The dependencies among these tiers are also factors to consider when tuning individual sub-systems.
- *Service dependency*
  A Web service that integrates with external services becomes dependent upon them. Poor performance of an external service can have a negative impact on the Web service.

Autonomic Computing [13] has emerged as a solution for dealing with the increasing complexity of managing and tuning computing environments. Computing systems that feature the following four characteristics are referred to as Autonomic Systems:

- *Self-configuring* - Define themselves on-the fly to adapt to a dynamically changing environment.
- *Self-healing* - Identify and fix the failed components without introducing apparent disruption.
- *Self-optimizing* - Achieve optimal performance by self-monitoring and self-tuning resources.
- *Self-protecting* - Protect themselves from attacks by managing user access, detecting intrusions and providing recovery capabilities.

Autonomic Computing will shift the responsibility for software management from the human administrator to the software system itself. It is expected that Autonomic Computing will result in significant improvements in terms of system management and many initiatives have begun to incorporate autonomic capabilities into software components.

One of these initiatives, proposed by Powley *et al.* [20], provides autonomic computing capabilities through the use of database functionality and reflective programming techniques. A reflective system maintains a model of self-representation and changes to the self-representation are automatically reflected in the underlying system. Reflection enables *inspection* and *adaptation* of systems at run-time [18] thus making reflection a viable approach to implanting autonomic features in computing systems. The Database Management System (DBMS) is used for data storage, creation of a knowledge base, and for controlled execution of logic flow in the system.

Autonomic Web Services Environment (*AWSE*) is a general framework for developing autonomic Web services [26]. It can be extended for use in any Service Oriented Architecture (SOA). Autonomic Web Services can also render efficient use of Web services for Enterprise Resource Management (ERM) [19], thus extending the use of Web services to areas other than e-business.

In this paper, we use the approach by Powley *et al.* to develop autonomic managers that are in turn used as the building blocks for an autonomic Web service based on the AWSE framework. The viability of our autonomic approach to Web service management is shown by the illustration of the self-tuning capabilities of a sample Web service to meet specified performance goals under changing workloads.

The remainder of the paper is structured as follows. Section 2 discusses the related work. The AWSE framework including the main concepts and technical details for building Autonomic Managers is described in Section 3. In Section 4 we validate our approach using a sample Web service implemented using the AWSE framework. Section 5 summarizes and concludes the paper.

2   RELATED WORK

SLA-based service management has been proposed by several researchers where different aspects of service management have been addressed. Dan *et al.* [9] propose a comprehensive a framework for SLA-based automated management for Web services with a resource provisioning scheme to provide different levels of service to different customers in terms of responsiveness, availability, and throughput. The customers are billed differentially according to their agreed service levels. The framework comprises the Web Service Level Agreement (WSLA) Language, a system to provision resources based on Service Level Objectives (SLO), a workload management system that prioritizes

requests according to the associated SLAs, and a system to monitor compliance with the SLA. Translation of WSLA specifications into system-level configuration information is performed by the service providers. Our research focuses on making each component in the Web service environment self-managing by augmenting each with an autonomic manager, and thereby, providing an overall QoS by the collaboration of the autonomic managers.

Levy *et al.* at IBM Research [17] propose an architecture and prototype implementation of a performance management system to provide resource provisioning and load balancing with server overload protection for cluster-based Web services. The system uses an inner level management for queuing and scheduling of request messages, and an outer level management for implementing a feedback control loop to periodically adjust the scheduling weights and server allocations of the inner level. It supports multiple classes of Web services traffic but requires users to first subscribe to services. While we address similar problems here, we use SLA based service provisioning rather than classification of service customers. Moreover, we use reflection and the facilities provided by a DBMS to implement the feedback control loop for autonomic management.

Sahai *et al.* [23] propose a Management Service Provider (MSP) model for remote or outsourced monitoring and control of E-services on the Internet. The model requires E-services to be instrumented with specific APIs to enable transaction monitoring using agent technology. An E-Service Manager is then deployed that manages the E-services remotely with the help of several other components. Sahai *et al.* [22] also propose an automated and distributed SLA monitoring engine for Web services using the Web Service Management Network (WSMN) Agent. Their approach uses proxy components attached to SOAP toolkits at each Web service site of a composite process to enable message tracking. WSMN Agents monitor the process flow defined using WSFL (Web Service Flow Language) to ensure SLA compliance. Our approach concentrates on management of individual Web service environments rather than using an agent framework for composite service management.

Specific service management issues have been addressed by different researchers. Fuente *et al.* [12] propose Reflective and Adaptable Web Service (RAWS), a Web service design model that is based on the concept of Reflective Programming. Source code maintenance can be done dynamically using RAWS without hindering the operation of the Web services. Web Service Offering Infrastructure (WSOI) [27] is proposed to demonstrate the usability of a dvWeb Service Offering Language (WSOL) in management and composition of Web services. Birman *et al*. [5] propose a framework that focuses

on reliable messaging, speedy recovery, and failure protection in order to implement high availability, fault tolerance and autonomous behavior for Web service systems. We propose a framework for managing all the components required to host a Web service and maintain an agreed-upon level of service performance. The self-healing, self-configuring and self-optimizing aspects of autonomic management are mainly addressed in our research work.

The Web Service Distributed Management (WSDM) [19] standard specification consists of two parts: Management Using Web Services (MUWS) defines how an IT resource is connected to a network and provides manageability interfaces to support local and remote control; Management of Web Services (MOWS) specifically addresses the management of the Web services endpoints using standard Web service protocols. By implementing Web service interfaces for the autonomic managers, the AWSE framework can be extended to demonstrate both MOWS and MUWS, and thereby, support the WSDM standard specifications.

The IBM Architectural Blueprint for Autonomic Computing [14] defines a general architecture for building autonomic systems. Diao *et al.* [10] identify a strong correspondence between many of the autonomic computing elements described in the blueprint and those in control systems and show how control theory techniques can be used for implementation of autonomic systems. Tzialas and Theodoulidis [28] also use control theory techniques for system control. However, their approach to building autonomic systems takes an ontological approach based on an extension of the Bunge ontology and the Bunge-Wand-Weber models. The ontological component models are captured using software engineering diagrams and the system is modeled as an organized whole, exhibiting a coherent overall behavior by controlling and managing the interactions between the components. The feedback control loop in our model uses the concept of reflection and is controlled using the features and functionality of a database management system.

Chung and Hollingsworth [7] propose an automated cluster-based Web service performance tuning infrastructure where single or multiple Active Harmony servers perform adaptive tuning using parameter replication and partitioning to speed up the tuning process. The paper also presents and evaluates a technique for resource sharing and distribution to allow Active Harmony to reconfigure the roles of specific nodes in the cluster. The self-tuning feature is implemented using a controller that implements optimization algorithms for determining the proper value of the tuning parameters. Bennani and Menascé [4] present self-managing computer systems by incorporating mechanisms for self-adjusting the configuration parameters so that the

QoS requirements of the system are constantly met. Their approach uses analytic performance models with combinatorial search techniques for designing controllers that run periodically to determine the best possible configuration for the system given its workload. We use policy guided automatic tuning to achieve specified performance goals.

Farrell and Kreger [11] propose a number of principles for the management of Web services including the separation of the management interface from the business interface, pushing core metric collection down to the Web services infrastructure. They use intermediate Web services that act as event collectors and managers. Our approach uses common tools provided by most database management systems as well as reflective programming techniques to incorporate self-awareness into the autonomic system.

3 AWSE FRAMEWORK

A Web services environment consists of multiple components including an HTTP server, application servers, database servers, and Web service applications. In our proposed architecture, we refer to a *Site* as a collection of components and resources necessary for hosting a Web service system provided by an organization (as shown in Fig. 1). Since these components may reside on multiple servers connected by network, a site can span multiple servers.

AWSE is an *Autonomic* Web Services Environment, that is, a system that is capable of self-management to ensure SLA compliance. SLAs are negotiated between a service consumer and the site's *SLA Negotiator*. The system automatically monitors performance and configures itself to ensure that the SLAs are met. This self-management is a continuous process whereby the system dynamically adapts to workload shifts and other changes in the environment.

In AWSE, we consider each component of the Web services environment to be autonomic, that is, self-aware and capable of self-configuration to maintain a specified level of performance. The core management software in the framework is the *Autonomic Manager* (as shown in Fig. 2) that manages an associated component such as the DBMS, or the HTTP server. The component, thus augmented with an autonomic manager, becomes an *Autonomic Element* that is capable of self-management. These autonomic elements form the basis of the AWSE framework and collaboration among the managers provides overall system management.

3.1 Key Concepts

The autonomic manager, the key building block of the AWSE framework, implements the
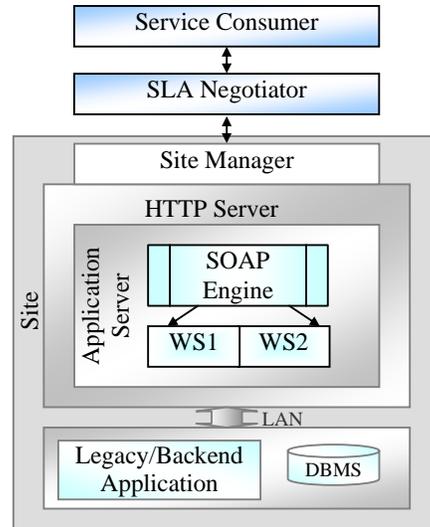

Figure 1: Web services hosting site

management capabilities of a resource, or a set of resources, called *managed elements*. A managed element may be any type of resource, hardware (e.g. storage units, servers) or software (e.g. DBMS, custom application), that is observable and controllable. Each managed element is augmented with an autonomic manager that performs the self-management tasks.

The autonomic manager, as described in the IBM Architectural Blueprint for Autonomic Computing (as shown in Fig. 2), typically consists of the following components: a Monitor (including sensors) that collects the performance data; an Analyzer that uses the performance data in light of system policies and goals to determine whether or not the system is performing properly; a Planner that determines, when necessary, appropriate actions to take; and the Executor (including effectors) that executes the suggested action(s) to control the managed element. The autonomic manager requires a storage facility to manage and to provide access to the system knowledge. Finally, it requires communication mechanisms and standard interfaces for the internal components to share information as well as for autonomic managers to communicate amongst themselves. The autonomic manager is typically implemented as a feedback control loop, sometimes referred to as the MAPE loop, encompassing the Monitor, Analyze, Plan and Execute components.

Central to all of the MAPE functions is knowledge about the system such as performance data reflecting past, present and expected performance, system topology, negotiated Service Level Agreements (SLAs), and policies and/or rules governing system behavior. Therefore, data management is an important aspect of the autonomic manager. A potentially large amount of data is collected, processed, stored, and queried by the different components of the autonomic manager.
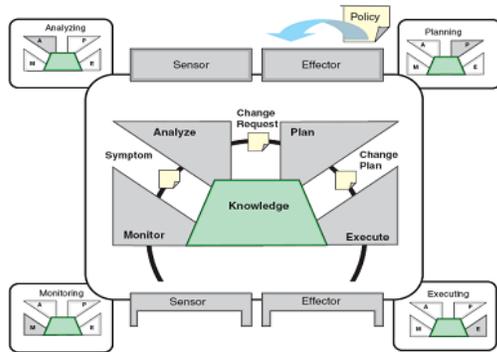
Figure 2: Autonomic Manager

These capabilities are obviously best provided by a database management system.

Our autonomic managers are built using a novel methodology which we call a *reflective and database-oriented* approach. Our approach relies heavily on common database management system (DBMS) concepts and tools such as triggers, user-defined functions and stored procedures. Not only does the DBMS store the system knowledge, performance data, policies and other information, it also serves as the central control of the autonomic manager, implementing the control flow logic for the manager. Database tables augmented with triggers that invoke stored procedures orchestrate the components of the MAPE loop. This reliance on the DBMS for storage and control is what makes our approach "database-oriented".

A reflective system is one that exposes a representation of its functional status allowing this representation to be inspected and modified while these modifications are "reflected" (effected) in the internal workings of the system [18]. In our approach, the self-representation of a component is stored in database tables. Triggers defined on the `Self-Representation` table invoke functions that effect the change within the corresponding component. In our approach, we consider the autonomic element's controllable features, along with their current status, to be the component's self-representation. When a configuration change is warranted, the self-representation table is updated. The update in turn fires a DBMS trigger that invokes a stored procedure that effects the configuration change within the component.

3.2 Autonomic Manager

The general reflective and database-oriented framework for building autonomic managers is applied to all autonomic managers in AWSE. Each component exposes a self-representation that can be modified either by the component itself, or by other autonomic managers. Each autonomic manager implements the MAPE loop via DBMS triggers, stored procedures and user defined functions. The managers can share a single DBMS or each can use

their own DBMS where knowledge such as policies and system topology are replicated across the DBMSs as necessary.

In the following subsections we describe our framework for the construction of autonomic managers and outline how the various components are implemented using our reflective and database-oriented approach. For illustrative purposes, we detail two examples of the autonomic managers employed in the AWSE environment, namely an autonomic manager for the buffer pool of a database management system, and an autonomic manager that controls the connection pool of database connections for a Web service.

The buffer pool area of a database management system (DBMS) is a key resource for performance in a database system. The DBMS buffer pool acts as a cache for all data requested from the database. Given the high cost of I/O accesses in a database system, it is important for the buffer pool to function as efficiently as possible, which means dynamically adapting to changing workloads to minimize physical I/O accesses. A simple autonomic manager for the DBMS buffer pool involves sizing the buffer pool to minimize physical I/Os.

For Web services that make use of a database system, establishing a connection with the DBMS can be very slow. Most commercial products such as WebLogic [31] and WebSphere [30], offer connection pools as an efficient solution to this problem. A connection pool is a named group of identical connections to a database that are created when starting up the Application Server. Web services *borrow* a connection from the pool, use it, and then return it to the pool. Besides solving the aforementioned performance issue, a connection pool also allows us to limit the number of connections to the DBMS to ensure overload protection and to allocate the connections among different workloads to offer differentiated QoS. We describe an autonomic manager that dynamically adjusts the number of connections dedicated to each type of workload to meet response time goals.

We describe the various components of the autonomic manager using the DBMS buffer pool and the connection pool managers as examples. The examples are simplified for the purpose of illustration and validation of the framework. Typically in a realistic system, the amount of data collected and stored would be far greater than illustrated in these simple examples. As such more complicated algorithms may be necessary for the self-tuning logic than the ones used in our experiments.

3.2.1   Knowledge

The MAPE loop requires knowledge about the system topology, performance metrics, component-based and system-wide policies, and the expectations, or system goals. Knowledge used by

the MAPE loop is stored in a set of database tables that can be accessed internally by the autonomic element, or externally by other autonomic managers via standard interfaces. For our example buffer pool autonomic manager, the basic system knowledge is represented by the set of tables as shown in Fig. 3. These tables include BP_Perf_Data (the performance data for the buffer pools), Self-Representation (the current configuration settings for parameters affecting the buffer pools), Analyzer_Result (the results from the analyzer), Goals (performance expectations for the buffer pools) and Policy (policies governing the buffer pools). Similar tables are defined to store the data relevant to the connection pool autonomic manager.

BP_Perf_Data
timestamp (PK)
numlogicalreads numphysicalreads datawrites indexwrites asynch_datawrites async_indexwrites asyncreads physicalreadtime physicalwritetime hit_rate

Analyzer_Result
manager_id (PK) **timestamp (PK)**
result

Self-Representation
manager_id (PK)
pool_size change_pg_thresh dlock_chk_time lock_list max_locks io_cleaners sort_heap goal

Policy
manager_id (PK) **policy_name (PK)**
policy_spec

Goal
manager_id (PK) **goal_type (PK)**
goal_value

Figure 3: Database tables for DBMS buffer pool autonomic manager

The performance data for a managed element is collected by sensors and is stored in database tables specifically defined to accommodate the data required by the managed element. The autonomic manager is most efficiently implemented by extracting a subset of relevant data from the set of collected data. This requires the identification of the problem indicators for the managed element, that is, identifying which pieces of data are most indicative of the root cause of a problem, and which data will be most affected by changes to the system. This data is typically identified by experts, or is discovered by way of experimentation.

For the buffer pool autonomic manager, the data that most reliably predicts and most accurately depicts potential problems related to buffer pool performance is stored in the table BP_Perf_Data (see Fig. 3). It includes information about the types of I/O required by the database (data versus index reads, physical versus logical reads, asynchronous versus synchronous I/O) and, most notably, the buffer pool hit rate (the likelihood of finding a requested page in the buffer pool) which is, in most cases, a good indicator of buffer pool performance.

The self-representation of the autonomic manager reflects the status of the controllable features of the autonomic element. This information is stored in the Self_Representation table. A number of DBMS configuration parameters are related to buffer pool usage including the buffer pool size, the size of the sort area, and the number of asynchronous processes that can be spawned for pre-fetching data or for writing dirty pages back to disk. These parameters are the controllable features of the buffer pool and modifying the values of these settings has an effect on the efficiency of the buffer pool. In our approach, these parameters and their current values form the self-representation of the DBMS buffer pool. The Self-Representation table is initially populated using an SQL query to the DBMS catalog tables that store the DBMS configuration information.

Goals and policies for the components are stored in the Goal and Policy tables respectively. Buffer pool goals are typically specified in terms of hit rate and/or response time goals. Policies describe the rules used to adjust buffer pool sizes. For example, in our simple DBMS buffer pool autonomic manager, a hit rate of less than 80% triggers an increase in the buffer pool size.

For the connection pool autonomic manager, the two metrics tracked and stored are rejection rate and response time. These metrics are indicative of the efficiency of the connection pool. The controllable feature of this manager, hence the self-representation, is the current number of connections to the database. The goal of the connection pool is expressed in terms of response time.

3.2.2   Monitor/Sensor

The implementation of the monitor for a managed element is dependent upon the type of interface and/or the instrumentation provided by the element. Commercial DBMSs typically provide several APIs and various monitors that can be used for monitoring the performance of the system [6]. The sensors are responsible for extracting relevant performance data for the managed element. Depending upon the autonomic element, the sensor may extract the data from log files, run monitoring tools associated with the element, or it may use an API to collect the data. The rate of data collection and the detail of collection may vary over time, depending upon the needs of the manager.

The monitor component of the MAPE loop is invoked by the sensors. It is used to filter and correlate sensor data and insert the data into one or more relational database tables specific to the element. Therefore, the monitor component must be customized to the needs of the autonomic manager.

We combined the monitor and the sensor as a single component to reduce the overhead. An application program represents the monitor/sensor

component in our implementation and uses the monitoring APIs for the DBMS to retrieve the data relevant to buffer pool performance. The program collects the data periodically, and since we use raw data for analysis, it simply inserts that raw data into the `BP_Perf_Data` table. The frequency of data collection is a parameter passed to the monitoring utility. The monitor/sensor for the connection pool autonomic manager retrieves the rejection rate and the response time by monitoring the requests to the Web service.

3.2.3 Analyzer/Planner

The analyzer and planner modules of the MAPE loop are custom built components specific to the autonomic element. The analyzer is responsible for examining the current state of the system and flagging potential problems. The planner module determines what action(s) should be taken to correct or circumvent a problem. These components may be policy-driven, or they may require complex logic and/or specialized algorithms.

In our example autonomic managers, the analyzer defines the logic to examine performance data in light of the defined policies and goals for the buffer pools and the connection pool, and determines whether or not the component is meeting its expectations as defined by the component goals.

The planner implements the algorithm(s) that define the adjustment(s) required to achieve the expectations for the managed element. For the buffer pool example, the planner simply increases the buffer pool size by 1000 4K pages to achieve better performance. For the connection pool example, the planner uses a *Queuing Network Model (QNM)* that incorporates a feed-forward approach to predict the number of connections which will satisfy the response time goals. In our approach, a multi-class Mean Value Analysis algorithm [16] is employed to estimate the workload response time for multiple classes of workloads. Both the number of concurrent requests and the think-time for each workload class are acquired from the workload monitor. When a workload changes, the planner performs an exhaustive search to generate the best allocation plan that meets the response time goals, and accordingly, the connections are re-allocated among the different classes of workloads. This is feasible in our case because of our small numbers of workload classes and connections pools. If search efficiency is required, a heuristic-based search [21] can be considered. The analyzer and the planner for both the connection pool and DBMS buffer pool autonomic managers are implemented as user defined functions and are thus stored in, and are accessible from within the DBMS.

3.2.4 Execute Module/Effectors

The concept of reflection is used in our approach to implement the effectors for an autonomic manager. The self-representation of the system embodies the current configuration settings for the managed element. These represent the features of the managed element that can be controlled. The self-representation for the DBMS buffer pools includes the current settings for tunable parameters such as the size of the buffer pool, the number of I/O pre-fetchers, and the number of I/O cleaners (threads to asynchronously write pages back to disk). The value of each parameter is adjustable and, when changed, affects the system performance. For the connection pool, the self-representation is simply the number of connections in the connection pool.

The self-representation information for a managed element is stored in the `Self_Representation` table. An update trigger on the value attribute in this table is used to implement the effectors, that is, the mechanisms that effect change to the managed element. When a change is made to the self-representation, the trigger initiates the execute module, which invokes the effector to make the actual configuration change(s). In the DBMS buffer pool example, the trigger calls external code that uses the DBMS API to change the buffer pool size.
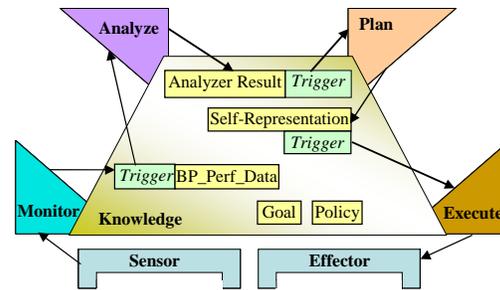

Figure 4: Autonomic Manager for Buffer Pools

3.2.5 Logic Flow

The MAPE loop is implemented as a feedback control loop that repeatedly monitors the component, analyzes its status, and makes necessary adjustments to maintain a predefined level of performance. The control of the feedback loop in our approach is largely implemented by database triggers defined on the database tables that store the system knowledge. The logic flow is shown in Fig. 4.

We describe the flow of control using the DBMS buffer pool autonomic manager beginning with the sensors. The sensors periodically collect data and pass it to the monitor which inserts this data into the performance data tables. An insert trigger is defined on the performance data table that invokes the analyzer module whenever new data arrives. Depending on the frequency of data collection the trigger may be modified to fire only periodically, as opposed to every time new data is inserted. The trigger defined on the `BP_Perf_Data` table is defined (using an IBM DB2 database) such as the following:

```
CREATE TRIGGER NewDataTGR
      AFTER INSERT ON BP_Perf_Data
      FOR EACH ROW MODE DB2SQL
      BEGIN ATOMIC VALUES (AnalyzeBPData())
```

The analyzer code may be implemented as a stored procedure, a user defined function, or as an external program that can be called by the database trigger. In our example, the analyzer code is defined as a user defined function called "AnalyzeBPData". If the analyzer detects a problem, it places a result or a notification in the `Analyzer_Result` table in the database. An insert trigger on the `Analyzer_Result` table calls the planner module whenever the result indicates that some action may be required.

The planner determines the appropriate action to take and updates the appropriate data in the `Self-Representation` table to initiate the reconfiguration. The update trigger on this table signals the execute module to take the suggested action. In the IBM blueprint, the execute module effects change to the monitored element by way of the effectors, that is, it makes changes to the configuration of the managed element. In our reflective approach, the planner makes a change to the managed element's self-representation and an update trigger defined on this table acts as the effector, the mechanism that actually makes the change to the managed element. The implementation of the routine to make the change depends upon the nature of the managed element. Changes may be implemented via the element's API, or they may involve updating configuration files and possibly restarting the component.

```
public interface Performance {
    // retrieves a list of data available for the component
    public Vector getMetaData();
    // retrieves the most recent performance data
    public Vector getCurrentData();
    // returns a specified part of most recent performance data
    public Vector getData(Vector params);
}
public interface Goal {
    // retrieves a list of goals that can be set for the component
    public Vector getMetaData();
    // retrieves the current goal for the component
    public Double getGoal (String goalType);
    // set a goal for the component
    public Boolean setGoal(String goalType, Double value)
}
```
Figure 5: Management Interface Specifications

3.2.6   Communication Interfaces

Two management interfaces, shown in Fig. 5, are defined for each autonomic element; the *Performance Interface* and the *Goal Interface*. The Performance Interface exposes methods to retrieve, query and update performance data. Each element exposes the same set of methods, but the actual data each provides varies. Meta-data methods allow the discovery of the type of data that is stored for each element. The Goal Interface provides methods to query and establish the goals for an autonomic
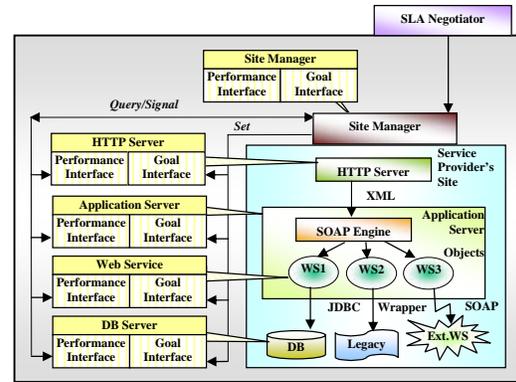

Figure 6: AWSE Architecture

element, thus allowing external management of a component. Meta-data methods promote the discovery of associated goals and additional methods allow the retrieval of current goals.

3.3 AWSE Management Hierarchy

As mentioned previously, AWSE is composed of components that are autonomic. However, although a component may appear to be performing well in isolation, it may not be functioning efficiently in an integrated environment with respect to the overall system objectives. An autonomic environment requires some control at the system level to achieve system-wide goals.

Each component (such as the HTTP server or the database server) in the AWSE framework may require one or more autonomic managers for monitoring service provisioning and controlling different parts of the component. Thus AWSE is composed of a hierarchy of communicating and cooperating autonomic managers. The higher level managers in the hierarchy query other managers at the lower levels to acquire current and past performance statistics, consolidate the data from various sources, and use pre-defined policies and goals to ensure satisfactory management of lower level managers. Our proposed architecture for AWSE, with the communication interfaces, is shown in Fig. 6. At the top level of the component manager hierarchy is the AWSE *Site manager* (the highest level autonomic manager in AWSE), which manages the overall performance of the site and ensures that SLAs are satisfied. It also provides external communication to allow remote and system-wide management of multiple sites within an organization.

Autonomic managers, therefore, must be able to communicate to share information. In our approach, communication is achieved via standard Web service interfaces exposed by the autonomic managers.

4   VALIDATION

We demonstrate two of the autonomic managers to illustrate the validity of our AWSE architecture;

one for the buffer pool of a database management system, and the other that controls the connection pool of database connections for a Web service. Our AWSE implementation prototype consists of a single site containing an HTTP server, an application server, a Web service application, and a DBMS that is accessed by the Web service. As described in Section 3, the negotiated SLAs are translated to system level goals that are monitored and managed by the Site manager. The Site manager primarily sets the goals for the autonomic managers of the components of the site such as, the HTTP server, Application server, Web services, and the DBMS. The autonomic managers of these components, in turn, set individual goals for the autonomic managers of their internal components. For example, the DBMS autonomic manager sets a goal for the buffer pool autonomic manager, and the Web service autonomic manager sets a goal for the connection pool autonomic manager.

The strategy for mapping of the SLAs to system level goals and distribution of the goals for the lower level autonomic managers is beyond the scope of this paper. These issues are currently being researched as future extensions to the AWSE framework. Our current implementation assumes predefined response time goals for the buffer pool and connection pool autonomic managers.

The Web service in our prototype AWSE provides two methods; one represents an Online Transaction Processing or OLTP-like workload (short transactions that retrieve small amounts of data), and the other represents an Online Analytical Processing or OLAP-like workload (longer running queries accessing large amounts of data). We call these two Web service methods *OLTP* and *OLAP* respectively. Each Web service method invocation queries data from a specific database table. A client application (running on another machine) simulates *users* to generate *interactive* workloads for the Web service according to a given ratio of OLTP/OLAP calls. "Workloads" referred to in our experiments are differentiated by the type of method invocation. Therefore, there are two workloads; an OLTP workload which is comprised of all calls to the OLTP Web service method, and an OLAP workload which corresponds to all calls to the OLAP method. Response time goals are defined individually for each workload. We assume that queries to the OLAP Web service method are more important than those to the OLTP method to resolve priorities in meeting the response time goals. In this section we show how the response time goals are achieved autonomically for the Web service connection pool and the database management system.

We use Apache HTTP server [2], Tomcat application server [3], an Axis-based Web service, and a DB2 Universal Database (UDB) [15] database server in our experimental setup. The HTTP server, application server, and Web service reside on a single machine (IBM Thinkcenter, 2.8Ghz, 1GB RAM) while the DBMS resides on a separate identical machine.

4.1 Connection Pool

Fifteen connections are created and allocated to the Web service upon initialization of our experimental environment. When a call is made to the Web service, a connection is allocated to the call if one is available; otherwise, the call is queued until a connection is available. As such, the number of connections allocated to the workload can have a significant impact on the service response time. An appropriate allocation of the connections among the workloads, therefore, is important to achieve response time goals. In our experiments, we define a 5000ms response time goal for the OLAP Web service method. The goal for the OLTP method is considered "best effort", so no specific goal is defined for this service.

The load on the Web service is varied dynamically by modifying the number of concurrent users making calls to each method. Fig. 7 shows the dynamically changing workload.

Initially, we test the case where all fifteen connections are shared, that is, no autonomic control is exerted. The results are shown in Fig. 8. In this case, we see that the defined response time goal for the OLAP Web service call is violated at some points due to the competition for connections between the two methods. We also observe that the response time of the OLAP service is more sensitive to the OLTP workload variations than it is to its own (comparing with the workload variation in Fig. 7), while the OLTP service call's response time is sensitive to increases in both types of workloads. The results show that 42% of the completed requests have response times greater than the 5000ms goal. On average, the requests violating the goal exceed it by approximately 20%.

To meet the response time goals, the autonomic manager allocates the fixed number of connections to different workloads to reduce the competition. Separate queues are used in a First Come First Served (FCFS) manner to buffer database requests of different workload classes.
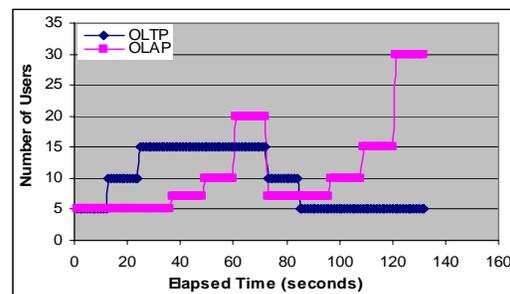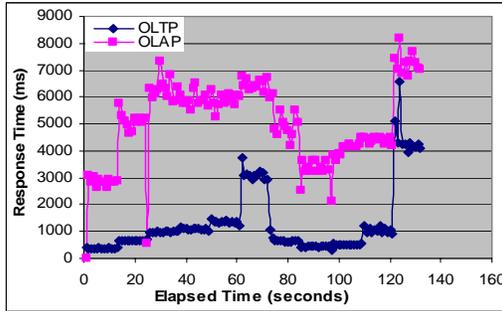

Figure 7: A dynamic workload

Figure 8: Shared connections

Figure 10: Adaptive response times

Using the workload shown in Fig. 7, we ran the experiment under the control of the autonomic manager. The connection allocation schemes used by the autonomic manager are shown in Fig. 9, which gives more importance to the OLAP workload as mentioned earlier in this section. We see that the number of connections allocated to OLAP Web service calls increases gradually in response to the increases in the OLAP workload. The resulting response times are shown in Fig. 10 . The surge of the OLTP response time between time 60 and 72 is caused by the heavily biased allocation of 14 connections to the OLAP workload and only 1 connection to the OLTP workload, during the presence of a higher workload of 20 OLAP users and 15 OLTP users. Such allocation is made due to the priority policy used by the autonomic manager to mitigate the violation of the response time goal of the OLAP service. We observe that the response time for the OLAP service hovers close to the response time goal of 5000ms up to time 120. Up to this time, 13% of the completed requests violated the goal, but this violation is less than 4% of the goal. After time 120, the system demonstrates an extreme allocation scheme where all the connections are allocated to the OLAP service. However, the response time of the OLAP service still remains far above the 5000ms goal. This is caused by the heavy OLAP workload of 30 users as shown in Fig. 7. Therefore, we can conclude that 15 connections are not sufficient to satisfy the requirements of such high (more than 85%) OLAP workload.
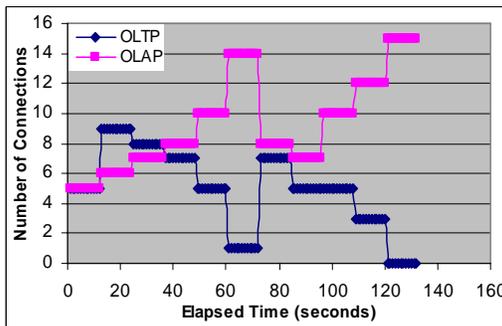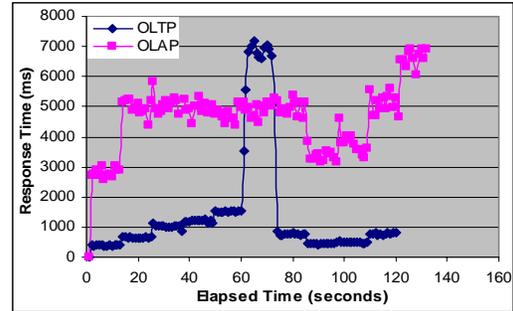
4.2 Buffer Pool

A single buffer pool is used to cache the database data for both the OLTP and the OLAP services. The goal for buffer pool is to maximize the hit rate, thus minimizing physical I/O. Given the nature of the OLAP Web service workload, which involves mainly sequential scans of the table, the hit rate will increase only when the entire table fits in the buffer pool. In this experiment, we begin with a pure OLAP workload and observe how the system adjusts the buffer pool size to achieve the response time goal. Once this goal has been reached, we introduce a second workload, the OLTP workload, which interferes with the OLAP performance. We expect that the system will adjust the buffer pool size further to accommodate both workloads. For the purpose of this experiment, we assume that there is sufficient memory in the system to satisfy the required buffer pool changes.

The OLAP service call scans a table that is close to 30MB in size, thus requiring approximately 7700 4K pages to accommodate the table in the buffer pool. We start with a buffer pool size of 1000 4K pages. The system monitors the hit rate every 20 seconds, and increments the buffer pool size by 1000 pages whenever the hit rate is found to be below 80%. Fig. 11 shows the autonomic adjustment of the buffer pool size and Fig. 12 shows the corresponding changes in buffer pool hit rates at different times corresponding to different buffer pool sizes. In Fig. 11, we observe that the buffer pool reaches 8000 4K pages, enough to hold the entire table in the buffer pool, at around time 140. Fig. 12 shows that around
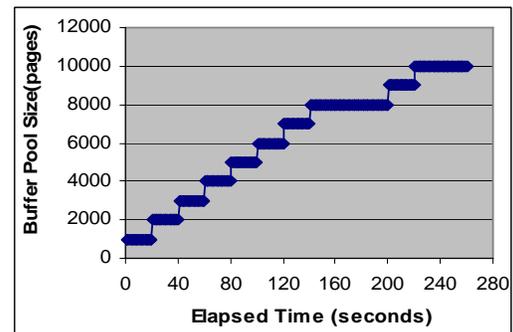

Figure 9: Connection allocations

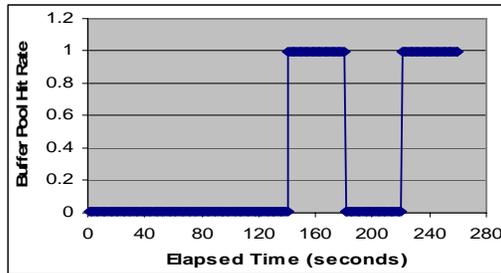**Figure 11:** Buffer pool size adjustments

Figure 12: Buffer pool hit rate adjustments

the same time (140s) the hit rate remains at 100% and no further adjustment is necessary at that point.

At time 180, we introduce the OLTP workload which competes with the existing OLAP workload for buffer pages. As a result, we observe that the hit rate dramatically drops to 0, and the tuning process is invoked again. The tuning process completes when the buffer pool reaches10000 pages, which is large enough to accommodate both tables.

Fig. 13 shows the resulting reductions in response times for the OLAP and OLTP service calls. The spikes are caused by the overhead of reconfiguring the buffer pool size.
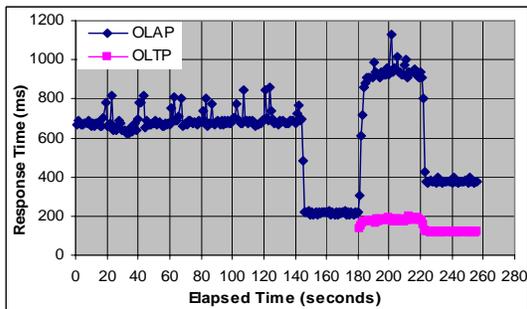

Figure 13: Web service response time with autonomic buffer pool adjustment

5   CONCLUSIONS

Autonomic systems are deemed indispensable for managing the growing complexity of information technology. Web services require an adaptive and autonomic management system to satisfy a widely varying workload and maintain agreed-upon service levels to leverage its usage in e-commerce. We propose AWSE, which builds on the IBM Architectural blueprint and provides a novel framework for autonomic Web services.

The core component of AWSE is an autonomic element, which is basically any component within the Web service hosting site, augmented with an autonomic manager, thus making the component self-managing. We present a design and implementation model for the autonomic manager as well as a coordination framework and the interfaces of autonomic managers for the management of a

Web service hosting site. Our approach exploits the powerful capabilities of a database management system for system control as well as for managing and storing the vast array of knowledge required for an autonomic system.

The experimental results presented in this paper show that goal-oriented reflection-based managers are effective in achieving autonomic resource management. The manager automatically controls an element's resources to meet the predefined individual and thereby, the system's overall performance goals. The self-managing components thus render a complete autonomic environment through the organization hierarchy and communication of the autonomic managers.

AWSE can extend the implementation of Web services more efficiently to areas such as Enterprise Resource Management (ERM) [19]**Error! Reference source not found.**, or Enterprise Project Management (EPM) [33], and network and systems management [34]. The general framework for the implementation of autonomic managers can be adapted to design more complex heterogenic systems by identifying the system components and leveraging them to autonomic elements.

Our current research focuses on implementing WSDM [19] standards in AWSE to enhance standard based communication among the autonomic managers. We are also working on the development of a site manager to control the hierarchy of managers and to perform goal distribution and assignment, which involves breaking down a high level system goal into appropriate component level goals for the hierarchical management in AWSE.

We also plan to extend the framework by incorporating SLA negotiation, automating Web service discovery, deriving appropriate models for different types of autonomic managers, and generating metrics with the performance data to generate QoS statistics to facilitate QoS based service discovery.

REFERENCES

[1] Akamai Technologies: The Impact of Web Performance on E-Retail Success. White paper, (2004).

[2] Apache HTTP Server, Retrieved from: http://httpd.apache.org/ (2007).

[3] Apache Tomcat, Retrieved from: http://tomcat.apache.org/ (2007).

[4] M. Bennani, and D. Menascé: Assessing the Robustness of Self-Managing Computer Systems under Highly Variable Workloads, In *Proc. of the Int. Conf. on Autonomic Computing (ICAC)*, NY, USA, (2004).

[5] K. Birman, R. Renesse, and W. Vogels: Adding High Availability and Autonomic Behavior to Web Services, In *Proc. of Int. Conf. on Software*

Engineering (ICSE), Scotland, UK (2004).

[6] P. Bruni, N. Harlock, M. Hong, and J. Webber: DB2 for z/OS and OS/390 Tools for Performance Management. *IBM Redbooks* (2001).

[7] I. Chung and J. Hollingsworth: Automated Cluster-based Web Service Performance Tuning. In *IEEE Conf. on High Performance Distributed Computing (HPDC),* Hawaii USA (2004).

[8] M. Clark, P. Fletcher, J. Hanson, R. Irani, M. Waterhouse, and J. Thelin: Web Services Business Strategies and Architectures, Wrox Press, (2002). Retrieved: Amazon.com (2007).

[9] A.Dan, D.Davis, R.Kearney, A.Keller, R.King, D.Kuebler, H.Ludwig, M.Polan, M.Spreitzer, and A.Youssef: Web Services on Demand: WSLA-driven automated management. *IBM Systems Journal*, Vol.43(1), pp. 136–158 (2004).

[10] Y. Diao, J.L. Hellerstein, G. Kaiser, S. Parekh, and D. Phung: Self-Managing Systems: A Control Theory Foundation. *IBM Research Report* RC23374 (W0410-080), (2004).

[11] J.A. Farrell, and H. Kreger: Web Services Management Approaches. *IBM Systems Journal*, Vol. 41(2), pp.212-227 (2002).

[12] J. Fuente, S. Alonso, O. Martínez, and L. Aguilar: RAWS: Reflective Engineering for Web Services, In *Proc. of IEEE Int. Conf. on Web Services (ICWS)*, San Diego, CA, USA (2004).

[13] A.G. Ganek, and T.A. Corbi: The Dawning of the Autonomic Computing Era, *IBM System Journal*, Vol. 42(1), pp.5-18 (2003).

[14] IBM. An Architectural Blueprint for Autonomic Computing, White Paper (2005).

[15] IBM DB2 Universal Database. At: http://www.software.ibm.com/data/db2/udb (2006)

[16] E.D. Lazowska, J. Zahorjan, G.S. Graham, and K.C. Sevcik: Quantitative System Performance, Computer System Analysis Using Queuing Network Models, by prentice-Hall, Inc. Englewood Cliffs, NJ (1984).

[17] R.Levy, J.Nagarajarao, G. Pacifici, M. Spreitzer, A.Tantawi, and A.Youssef: Performance Management for Cluster-based Web Services, In *proc. of IFIP/IEEE Int. Symposium on Integrated Network Management (IM'03)*, Colorado Springs, USA, pp. 247-261 (2003).

[18] P. Maes: Computational Reflection. *The Knowledge Engineering Review*, Vol. 3(1), pp. 1-19, (1988).

[19] OASIS: An Introduction to WSDM. Committee Draft, 2006. Retrieved from: http://www.oasis-open.org/committees/download.php/16998/wsdm-1.0-intro-primer-cd-01.doc (2006).

[20] W. Powley and P. Martin: A Reflective Database-Oriented Framework for Autonomic Managers. In *Proc. of Int. Conf. on Networking and Services (ICNS),* San Jose, CA, USA(2006).

[21] V.J. Rayward-Smith, I.H. Osman, C.R. Reeves, Smith, G.D. (Eds.): Modern Heuristic Search Methods, John Wiley and Sons, Canada (1996).

[22] A. Sahai, V. Machiraju, M. Sayal, A. Van Moorsel, and F. Casati: Automated SLA Monitoring for Web Services, In *Proc. of the IFIP/IEEE Int. Workshop on Distributed Systems: Operations and Management (DSOM'02),* Montreal, Canada. LNCS, Springer, Vol. 2506, pp. 28-41 (2002).

[23] A. Sahai, V. Machiraju, and K. Wurster: Monitoring and Controlling Internet based Services, In *Proc. of IEEE Workshop on Internet Applications (WIAPP)*, San Jose, CA (2001).

[24] SOAP Ver. 1.2 Part 1: Messaging Framework, At: http://www.w3.org/TR/soap12-part1/ (2007).

[25] M. Tian, T. Voigt, T. Naumowicz, H. Ritter, and J. Schiller: Performance Impact of Web Services on Internet Servers, In *Proc. of Int. Conf. on Parallel and Distributed Computing and Systems (PDCS)*, Marina Del Rey, USA (2003).

[26] W. Tian, F. Zulkernine, J. Zebedee, W. Powley, and P. Martin: Architecture for an Autonomic Web Services Environment. In *Proc. of the Joint Workshop on Web Services and Model-Driven Enterprise Information Systems (WSMDEIS),* Miami, Florida, USA (2005).

[27] V. Tosic, W. Ma, B. Pagurek, and B. Esfandiari: Web Services Offerings Infrastructure (WSOI)-A Management Infrastructure for XML Web Services, In *IEEE/IFIP Network Operations & Management Symposium (NOMS),* Seoul, South Korea, pp. 817-830 (2004).

[28] G. Tziallas and B. Theodoulidis: Building Autonomic Computing Systems Based on Ontological Component Models and a Controller Synthesis Algorithm. In *Proc. of the Int. Workshop on DEXA*, Prague, Czech Republic, pp. 674-680 (2003).

[29] UDDI Version 3.0.1, UDDI Spec Technical Committee Specification, 2003. At: http://uddi.org/pubs/uddi_v3.htm, (2007).

[30] WebSphere:http://demo.freshwater.com/SiteScope/docs/WebSphereMon.htm. (2007).

[31] WebLogic:http://edocs.bea.com/wls/docs81/perform/index.html (2007).

[32] Web Services Description Language (WSDL) 1.1 At: http://www.w3.org/TR/wsdl.html (2007).

[33] L. Zhang, H. Cai, J. Chung, and H. Chang: WS-EPM: Web Services for Enterprise Project Management, In *Proc. of the IEEE Int. Conf. on Services Computing (SCC'04)*, Shanghai, China, pp. 177-185 (2004).

[34] L. Zhang and M. Jeckle: Web Services for Integrated Management: A Case Study. In *Proc. of Web Services: European Conf. (ECOWS)*, Erfurt, Germany (2004).