Architecture for an Autonomic Web Services Environment

Wenhu Tian, Farhana Zulkernine, Jared Zebedee, Wendy Powley, Pat Martin
School of Computing,
Queen's University, Kingston, ON Canada
{tian, farhana, zebedee, wendy, martin }@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 is, 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 an architecture of an autonomic Web service environment based on reflective programming techniques, where components at a Web service hosting site tunes themselves and collaborate to provide a self-managed and self-optimized system.

1. Introduction

Web services are self-contained and self-describing software components that can be accessed over the Internet. They are now well accepted in Enterprise Application Integration (EAI) [19] and Business to Business Integration (B2Bi) [4]. Performance plays a crucial role in promoting the acceptance and widespread usage of Web services. Poor performance (e.g. long response time) means the loss of customers and revenue [14]. 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, Web service performance is of utmost importance, and recently has gained a considerable amount of attention [3, 15, 18].

A Web service is a Web-accessible program that is described in a WSDL (Web Service Description Language) [17] document. Web services are published or discovered via a UDDI (Universal Description, Discovery and Integration) [16] registry. SOAP (Simple Object Access Protocol) [13] 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 in an ever-changing environment, these tasks must be automated.

Unacceptable Web service performance results from both networking and server-side issues [10]. 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 [7] 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.

In this paper we propose an architecture for an autonomic Web services environment. We consider each component in the proposed architecture as self-managing and thereby present a hierarchical layout of autonomic managers that constitute a self-configuring and self-optimizing autonomic Web service system. The remainder of the paper is structured as follows. Section 2 discusses related approaches to Web service management. Our proposed autonomic architecture is presented in Section 3, and a detailed scenario to illustrate how the architecture works is provided in Section 4. Section 5 summarizes and concludes the paper.

2. Related Work

Architectural approaches based on SLA-driven Web services have been proposed by Dan et al. [5] and Levy et al. [9]. Dan's framework includes components for the support of an SLA throughout its entire life-cycle as well as SLA-driven management of services. Levy et al uses a queuing model to predict response times for different resource allocations. In their model, the management system is transparent and allocates server resources dynamically to maximize the expected value of a given cluster utility function. Both of these approaches focus on service provisioning. We focus on autonomic management rather than the provisioning aspects.

Farrell and Kreger [6] 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. We incorporate these ideas and expand upon them in our approach.

The insufficient reliability and lack of autonomic features in current Web services architectures is presented by Birman et al in [2]. He proposes some extensions to the current Web services framework in the form of more robust monitoring and reliable messaging to achieve higher availability.

3. Autonomic Web Services Architecture

A Web services environment typically consists of a collection of components including HTTP servers, application servers, database servers, and Web service applications. In our proposed architecture, as shown in Figure 1, we consider each component to be autonomic, that is, self-aware and capable of self-configuration to maintain a specified level of performance. System-wide management of the Web services environment is facilitated by a hierarchy of *Autonomic Managers* that query other managers at the lower level to acquire current and past performance statistics, consolidate the data from various sources, and use pre-defined policies and SLAs to assist in system-wide tuning.

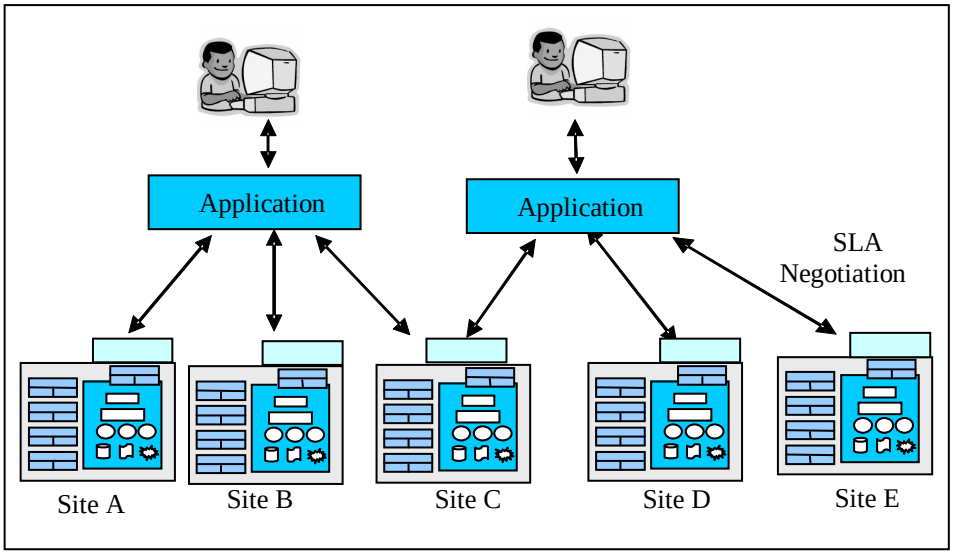

Fig. 1. Autonomic Web Services Architecture

At the lowest level in our architectural hierarchy are the Au*tonomic Elements*. We refer to an *autonomic element* as a component augmented with self-managing capabilities. An autonomic element is capable of monitoring the performance of its component, or *managed element*, (such as a DBMS or an HTTP server), analyzing its performance and, if required, proposing and implementing a plan for reconfiguration of the managed element. Autonomic elements form the building blocks of our architecture and are described in more detail in Section 3.1.

We refer to a *Site* as a collection of components and resources necessary for hosting a Web service system provided by an organization. A Web services hosting site typically consists of HTTP servers, application servers, SOAP Engines, and Web services. Web services are basically Web accessible interfaces or applications that can connect to other backend applications such as legacy systems, or database management systems. Most often these backend components are located on separate servers that are connected by a Local Area Network (LAN). A site can therefore span multiple servers. A *site manager* oversees the overall performance of the site and provides service provisioning for the components associated with the site.

An *Application*, as shown in Figure 1, is a special purpose client program that uses one or more Web services, possibly from different sites. An investor application, for example, that allows users to look up stock prices may use Web services from several different companies. A site's *SLA Negotiator* negotiates SLA agreements between the applications and the Web services hosted by the site. Once SLA agreements are made, the site must manage its resources to ensure the agreed level of performance.

There are two levels of management in our approach; the component level and the site level. The component is responsible for managing its own performance to meet goals specified by the site manager. The site manager monitors for SLA compliance, sets component goals, and provides resource provisioning when necessary.

3.1 Autonomic Elements

An autonomic element can be viewed as a feedback control loop as shown in Figure 2 [8], controlled by an *Autonomic Manager*. The autonomic manager oversees the monitoring of the component (the *Managed Element*), and by analyzing the collected statistics in light of known policies and goals, it determines whether or not the component performance is adequate. If necessary, a plan for reconfiguration is generated and executed.

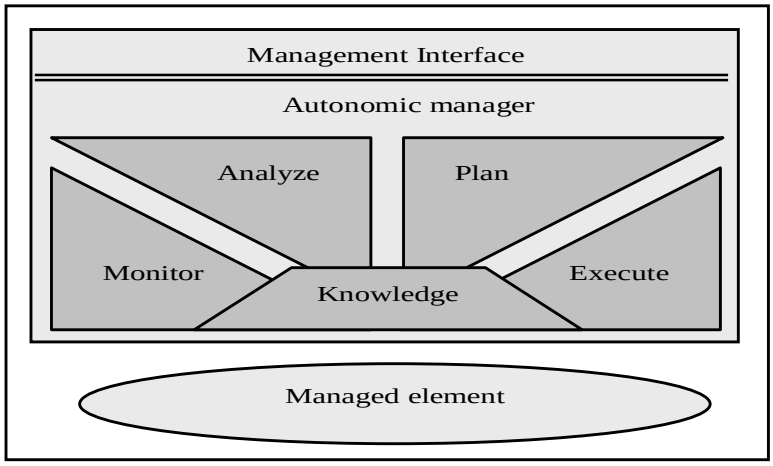

Fig. 2. Autonomic Element

One approach to building autonomic elements is based on the principles of *reflective programming* [11]. A reflective system is one that can inspect and adapt its internal behaviour in response to changing conditions. Typically a reflective system maintains a model of self-representation, and changes to the self-representation are automatically reflected in the underlying system.

An example of an autonomic database management system (DBMS) based on reflective programming techniques, was presented by Martin et al [12]. In this system, the self-representation of the system embodies the current configuration settings and the statistics that are collected regarding the system performance. This information is stored as a set of database relations that can be queried and updated. A monitoring tool periodically takes snapshots of the DBMS performance and stores the collected data in a data warehouse. When a new set of performance data is inserted into the data warehouse, a database trigger is fired that calls a diagnosis function. The diagnosis function compares current and past performance data to determine whether or not a change in configuration is warranted based on a preset desired performance setting. If one or more configuration parameters should be altered, a change is made to the self-representation which in turn triggers a change to the underlying DBMS configuration parameters.

We use this notion of reflection to implement Web components as autonomic elements. In our architecture, all components such as the HTTP server, the application server, the Web services and supporting applications as well as the site manager are instances of autonomic elements. Each component has an autonomic manager as shown in Figure 2, augmented with a reflective *Management Interface*. This interface is used by higher level managers to set performance goals as per Service Level Agreements (SLAs) for the managed element and to obtain current performance statistics for the component. As in the example of the autonomic DBMS, a monitoring tool periodically monitors the system performance and the analyzer compares the current and past performance to determine whether a configuration change is necessary to achieve the desired goal. Following the principles of reflective systems, each autonomic element maintains a self-representation which embodies the component's current goal settings and its current performance statistics. Updates made to the self-representation trigger changes to the actual system. If deemed necessary by the analyzer, changes are made to the self-representation to reconfigure the component.

In our proposed architecture, to ensure interoperability between autonomic elements, a common management interface is specified for all elements to provide access to the self-representation. Each autonomic element monitors itself to assess its general health and the performance data is stored as part of

the component's self-representation. This data can be accessed using the methods provided by the management interface. Historical data may be used for performance analysis and prediction.

The standard Web services environment already provides the tools required to define, publish, discover, and to use APIs across platforms. These tools and methods are exploited in our proposed architecture for communication between elements. To implement the reflective interface, we view each component as a Web service where the self-representation is accessed via Web service operations for each element. Two management interfaces 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

```
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)
}

public interface Performance{
       // retrieves a list of goals that can be set for the component
       public Vector getMetaData();
       // retrieves the most recent performance data
       public Vector getCurrentData();
       // returns a specified portion of the most recent performance data
       public Vector getData(Vector params);
}
```

Fig. 3. Management Interface Specifications

The Goal Interface provides methods to query and establish the goals for an autonomic element. Meta-data methods promote the discovery of associated goals and additional methods allow the retrieval of current goals. Goals for individual components can be set only by their associated site manager. Goals for a site manager are set by the site's SLA Negotiator component.

Component-level performance interfaces are accessed only by their associated site manager. A site manager uses the performance interface to asses the current health of each of its components and uses the component's goal interface to set individual goals for each component.

Management interfaces are defined and published using WSDL and a private management UDDI registry as suggested by Farrell and Kreger [6]. The self-representation can be stored using any storage format (database, log files etc) as these details are made transparent by the use of a Web service interface. Figure 3 shows the interface specification of the management interfaces common to all autonomic elements. The WSDL specification for the *setGoal()* method is given in the Appendix as an example.

Each autonomic element implements a monitoring component to asses the health of its managed element. Monitoring incurs a certain degree of overhead, so monitoring processes must be lightweight and invoked as infrequently as possible. Multiple levels of monitoring allow more information to be collected depending on the amount of detail that is desired. In some cases, it may be desirable to *drill down*, collecting more detailed information to assist in problem determination. At times of stable, acceptable performance, it may suffice to collect data less frequently.

Current HTTP servers and application servers provide rich interfaces for monitoring tools to extract performance statistics and running status. A variety of monitor tools are available on the market to visualize and analyze collected statistics, and if necessary, to fire warnings when the pre-set thresholds are violated [20, 1]. DBMSs are rich in monitoring tools and APIs for gathering information. Monitors can be switched on or off at will, and different levels of monitoring can be specified. Monitoring individual Web services presents more of a challenge as each Web service application is unique. Generic monitors can be

developed that provide basic information such as response time for the Web service, number of requests per time unit, or average queue length.

3.2 Site Management

A *site* is a collection of Web service components and resources provided by an organization that offers one or more Web services. The components comprising a site are shown in Figure 4. A site may be distributed across many physical nodes. Multiple instances of a component may reside on the same site and resources are provisioned as required.

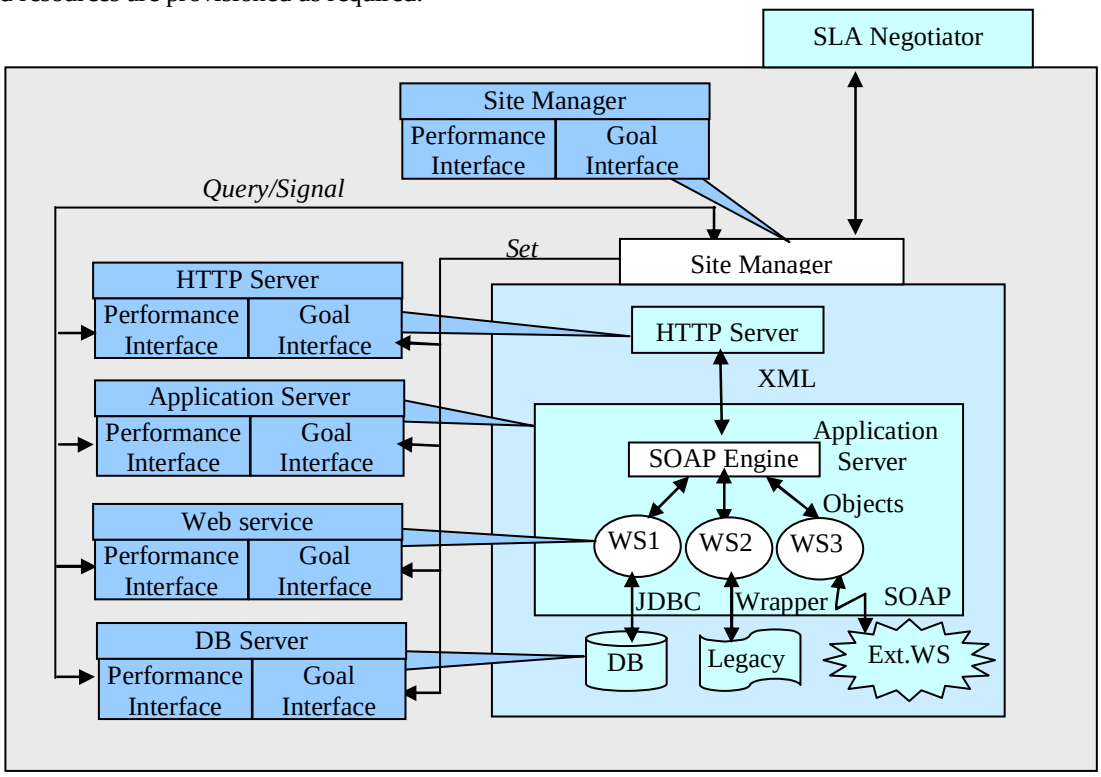

Fig. 4. Autonomic Web Services Site

Applications that wish to use the Web services offered by a site negotiate a SLA with the site's SLA Negotiator. Details of an automated approach to SLA negotiation is presented by Dan et al in [5], and is beyond the scope of this paper. We assume that different SLAs can be specified for each Web service or, if a finer level of granularity is required, SLAs can be set on a per-operation level. The site's SLA Negotiator translates these high level specifications into performance goals such as response time or average throughput for each Web service or operation. The SLA Negotiator component sets the goals for the site using the site's management interface.

Each site employs a *Site Manager* that oversees the general performance of the components comprising the site. The site manager itself is implemented as an autonomic element with its own autonomic manager. Conceptually, the site manager is the autonomic manager of all the components within the scope of the site. The site manager collects the performance statistics of each component by querying the management interfaces of the individual components. This information, along with the policies and goals defined for the site, is used to determine whether or not the performance of the site is adequate. If the site is in violation of one or more of the SLA agreements, an action plan is generated and executed. An action plan may involve the generation and setting of new goals for particular components, or it may involve a modification in the provisioning of resources.

The site manager is implemented as a Web service that exposes the site's performance interface that can be accessed by other site managers or external components. This interface can be used by applications for

error tracking, Web service selection, or by modules handling external SLA compliance monitoring. The performance data for a site provides summary data indicating the overall performance of the associated components.

The site manager is responsible for monitoring the overall performance of the Web services offered by the site. The site manager retrieves the performance data via the components' performance interfaces. The information required by the component for self-management may differ from that required for overall system management by managers at the site level. For instance, a DBMS focuses on low level resources such as I/O and CPU usage to maximize performance. To optimize site performance, and to monitor SLA compliance, the site manger requires higher level statistics such as throughput or transaction response times. This information is available through the components management interface.

4. Scenario

Functionality of the different components presented in the architecture of autonomic Web services system can be better explained using a common example like the Stock Quote composite Web service system shown in Fig. 5. In this system, a customer uses an *Investor* application to find out the details about multiple stocks. The *Investor* application invokes a *Stock Broker (SB)* Web service by sending a *register* message containing a list of stock IDs. The Stock Broker sends *accept* or *reject* message to the Investor in response. In case of *accept*, the Stock Broker sends the stock IDs received from the customer, one by one to the *Research Department (RD)* Web service. The RD finds the necessary information and sends a *report* directly to the Investor application. When the Investor receives information about all the stocks, it sends an *acknowledgement* message to the Stock Broker service. The Stock Broker service then submits the *bill* to the Investor and notifies the Research Department about the end of the job. The messages interchanged in this system are presented in Figure 5.

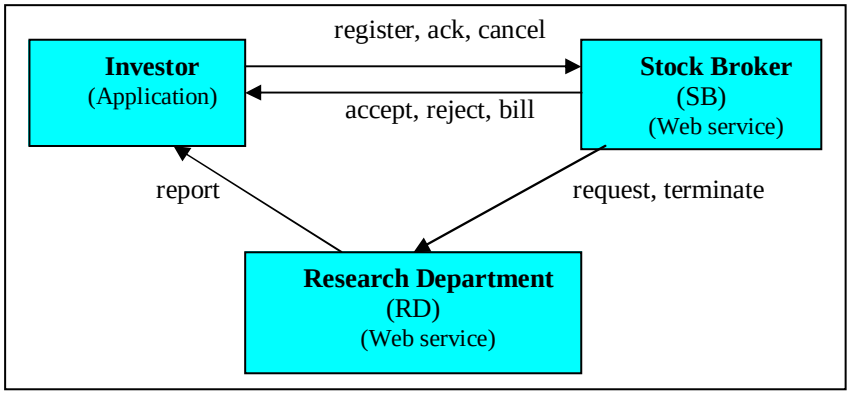

Fig. 5. Stock Broker Web Service System

The Stock Broker and Research Department Web services are located at two different sites. Each of these sites is managed by a site manager. The site manager receives the SLA from the SLA negotiator and monitors the performance of the different components at the site to provide an overall performance in compliance with the SLA. For the Stock Broker service system, the site manager monitors the performances of the HTTP server, application server, and other components at the site including the Stock Broker service.

If the SLA between the Investor and the Stock Broker site is in violation, the Stock Broker's site manager retrieves the performance data of all the individual components associated with this site, analyzes them, and sets new goals for the necessary components in order to avoid violation of the SLA. For example, if the maximum response time specified in the SLA is five seconds, and the observed response time is close to, or beyond this threshold, the site manager tries to set new goals for specific components to

reduce the response time to five seconds or less. If the perceived bottleneck is the HTTP server, the site manager uses the HTTP server's goal interface to set a new goal for this component.

Each component in the autonomic Web service system is associated with its own autonomic manager. When new performance goals are set, the specific components attempt to reconfigure themselves using their own autonomic managers. In our example, the HTTP server's autonomic manager may increase the number of threads to improve its response time.

At the highest level, the client Investor application sets the SLA for the Stock Broker service through the SLA negotiator before invoking the service. The SLA negotiator conveys the same to the Stock Broker's site manager and also to the linked services, in this case the Research Department. When all the linked services agree to the SLA, the Investor application can invoke the Stock Broker service. Both the application and the site manager monitor the service performance to ensure SLA compliance. For linked services, the site manager of the calling service does the monitoring while the SLA negotiator plays the role of the application in doing the SLA negotiation with the linked services.

5. Summary

Performance plays a crucial role in the eventual acceptance and widespread adoption of the Web services model of application deployment. Web service performance, however, is difficult to manage because of the complexity of the components and their interactions, and the variability in demand and the environment. In this paper, we propose autonomic computing as a solution to the problems in managing Web service performance. We describe an architecture for an autonomic Web services environment where each component is fully autonomic and equipped to cooperate in a managed environment. Each component provides a management interface that exposes a self-representation consisting of performance statistics and goal information. Our architecture uses standard Web service tools and protocols; interface definitions specified using WSDL and communication using SOAP over HTTP. Site level managers oversee the overall performance of the components and ensure SLA compliance.

We see that progress must be made in several areas before an autonomic Web services architecture, such as the one described in this paper, can be deployed. First, Web service components are currently not, for the most part, autonomic. In fact, in many cases, components require a complete shut-down and restart before configuration changes take effect, thus causing an interruption of service. Dynamic reconfiguration support is necessary for components to fit into an autonomic environment. As part of our research we are modifying open source Web based components, such as the Apache HTTP server, to enable dynamic configuration. Second, autonomic systems will require extensive monitoring, analysis and diagnosis. Most Web components currently provide sophisticated support to accomplish these tasks, however, ensuring that these processes do not burden the system with excessive overhead costs will be a challenge. Third, an architecture like the one proposed here relies on the specification of SLAs, goals and policies to determine acceptable performance. Users require a specification language in which these high level SLAs and policies can be expressed and SLAs must be translated into observable measures to be used as goals for each component. We plan to use the WSLA language [5] as the starting point and investigate how goals for individual components can be specified and derived from Web service SLAs.

References

1. Apache Server Monitor, http://demo.freshwater.com/SiteScope/docs/ApacheServerMon.htm.
2. Birman, K., van Renesse, R., and Vogels, W.: Adding High Availability and Autonomic Behavior to Web Services, *26th International Conference on Software Engineering (ICSE'04)*, May 2004, Edinburgh, Scotland, United Kingdom.
3. Chiu, K., Web Services Performance: A Survey of Issues and Solutions, *7th World Multiconference on Systemics, Cybernetics and Informatics (SCI 2003)*, Orlando, USA, July, (2003).
4. Fletcher, P., Waterhouse, M. (Eds).: Web Services Business Strategies and Architectures, Expert Press, (2002).

5. Dan, A., Davis, D., Kearney, R., Keller, A., King, R ., Kuebler, D., Ludwig, H., Polan, M., Spreitzer, M. and Youssef, A.: Web Services on Demand: WSLA-driven automated management. *IBM Systems Journal*, 43(1), (2004) 136 – 158.
6. J. A. Farrell, H. Kreger, Web Services Management Approaches. *IBM Systems Journal*, 41(2), (2002).
7. Ganek, A.G., Corbi, T.A.: The Dawning of the Autonomic Computing Era, *IBM System Journal*, V(42), N(1), (2003).
8. Kephart, J.O., Chess, D.M.: The Vision of Autonomic Computing. *Computer*, 36(1), (2003), 41-50.
9. Levy, R., Nagarajarao, J., Pacifici, G., Spreitzer, M., Tantawi, A.N., Youssef, A.: Performance Management for Cluster Based Web Services, *IFIP/IEEE 8th International Symposium on Integrated Network Management (IM 2003)*, (2003), 247-261.
10. Loosley, C., Gimarc, R.L., Spellmann, A.C.: E-Commerce Response Time: a Reference Model, *Keynote Systems Inc*., (2000).
11. Maes, P., Computational Reflection, *The Knowledge Engineering Review*, pp. 1-19, (1988).
12. Martin, P., Powley, W., Benoit, D.. Using Reflection to Introduce Self-Tuning Technology into DBMSs. *Proceedings of IDEAS'04*, Coimbra, Portugal, July 2004.
13. SOAP Version 1.2 Part 1: Messaging Framework, June 2004, http://www.w3.org/TR/soap12-part1/.
14. The Impact of Web Performance on E-Retail Success, Akamai Technologies, Feb. 1, (2004), http://www.akamai.com/en/resources/pdf/whitepapers/Akamai_eRetail_Success_Whitepaper.pdf.
15. Tian, M., Voigt, T., Naumowicz, T., Ritter, H., and Schiller, J.: Performance Impact of Web Services on Internet Servers, *International Conference on Parallel and Distributed Computing and Systems (PDCS 2003)*, Marina Del Rey, USA, (Nov. 2003).
16. UDDI Version 3.0.1, UDDI Spec Technical Committee Specification, (Oct. 2003), http://uddi.org/pubs/uddi_v3.htm.
17. Web Services Description Language (WSDL) 1.1, (Mar. 2001), http://www.w3.org/TR/wsdl.
18. Weikum, G.: Self-tuning E-services: from Wishful Thinking to Viable Engineering, *High Performance Transaction Systems Workshop Submissions*, (Oct. 2001).
19. Wong, S.: Web services: The Next Evolution of Application Integration, http://www.eaiindustry.org/docs/WebServicesTheNextEvolutionofApplicationIntegration.pdf.
20. WebSphere Application Server Monitor, http://demo.freshwater.com/SiteScope/docs/WebSphereMon.htm.

Appendix: WSDL Sample

The following shows the WSDL generated for the *setGoal* routine which is part of the Performance management interface.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://DefaultNamespace"
xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:apachesoap="http://xml.apache.org/xml-soap"
xmlns:impl="http://DefaultNamespace"
xmlns:intf="http://DefaultNamespace"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <wsdl:message name="setGoalResponse">

      <wsdl:part name="setGoalReturn" type="xsd:boolean"/>
   </wsdl:message>
   <wsdl:message name="setGoalRequest">
      <wsdl:part name="in0" type="xsd:string"/>
      <wsdl:part name="in1" type="xsd:double"/>
```

```
   </wsdl:message>
   <wsdl:portType name="Config">
      <wsdl:operation name="setGoal" parameterOrder="in0 in1">
         <wsdl:input message="impl:setGoalRequest"
name="setGoalRequest"/>
         <wsdl:output message="impl:setGoalResponse"
name="setGoalResponse"/>
      </wsdl:operation>
   </wsdl:portType>
   <wsdl:binding name="ConfigSoapBinding" type="impl:Config">
      <wsdlsoap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/>
      <wsdl:operation name="setGoal">
         <wsdlsoap:operation soapAction=""/>
         <wsdl:input name="setGoalRequest">
            <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://DefaultNamespace" use="encoded"/>
         </wsdl:input>
         <wsdl:output name="setGoalResponse">
            <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://DefaultNamespace" use="encoded"/>
         </wsdl:output>
      </wsdl:operation>
   </wsdl:binding>
   <wsdl:service name="ConfigService">
      <wsdl:port binding="impl:ConfigSoapBinding" name="Config">
         <wsdlsoap:address
location="http://webs2/axis/services/Config"/>
      </wsdl:port>
   </wsdl:service>
</wsdl:definitions>
```