An Extensible, Lightweight Architecture for Adaptive J2EE Applications

Ian Gorton
Pacific Northwest National Laboratory
WA, U.S.A
Ian.gorton@pnl.gov

Yan Liu
National ICT Australia
Australia
Jenny.liu@nicta.com.aul

Nihar Trivedi
School of Information Technologies,
University of Sydney, Australia
nihar.trivedi@nicta.com.au

ABSTRACT
Server applications with adaptive behaviors can adapt their functionality in response to environmental changes, and significantly reduce the on-going costs of system deployment and administration. However, developing adaptive server applications is challenging due to the complexity of server technologies and highly dynamic application environments. This paper presents an architecture framework, known as the *Adaptive Server Framework* (ASF). ASF provides a clear separation between the implementation of adaptive behaviors and the server application business logic. This means a server application can be cost effectively extended with programmable adaptive features through the definition and implementation of control components defined in ASF. Furthermore, ASF is a lightweight architecture in that it incurs low CPU overhead and memory usage. We demonstrate the effectiveness of ASF through a case study, in which a server application dynamically determines the resolution and quality to scale an image based on the load of the server and network connection speed. The experimental evaluation demonstrates the performance gains possible by adaptive behaviors and the low overhead introduced by ASF.

Categories and Subject Descriptors
C.2.4 [**Computer-Communication Networks**]: Distributed Systems - distributed applications; D.2.11 [**Software Engineering**]: Software Architecture - domain-specific architectures, patterns.

General Terms
Design.

Keywords
Adaptation, J2EE, component, software architecture.

1 INTRODUCTION
Autonomic computing encompasses a grand vision of self-managing, self-tuning application server technologies [9]. Applications server platforms are key elements of autonomic computing [9], providing a standardized service layer to support the development and deployment of distributed, server-side applications. Application server technologies such as J2EE and .NET can be used to build server applications that support various levels of quality-of-service, making them suitable platforms for high performance and reliable software systems.

However, server applications remain challenging to construct. This is due to the complexity of application server technologies and the fact that server applications operate in dynamic environments with variable request loads, fluctuating resource usage and unpredictable system faults. It is therefore difficult, if not impossible, to optimize the quality goals (such as performance, security, reliability and predictability) of an application in all circumstances.

Platforms for developing adaptive server applications exist [2] [14]. They typically require the application to be developed using custom middleware with specialized APIs and languages. This makes it infeasible to use these platforms to augment existing applications with adaptive behavior.

Another approach is to extend existing application server platforms with adaptive mechanisms [4]. By augmenting existing middleware, it becomes possible to transparently build adaptive capabilities into existing applications. It would also promote an attractive adoption path for new adaptive applications.

However, developing flexible and efficient adaptive control logic for server applications is challenging for at least three reasons. First, the ability to handle dynamic control must be addressed. Adaptation must occur at runtime in response to changes in the application's environment, because changing the system behavior at compile or load time is not practical. Second, overheads introduced by external control mechanisms must be low, so that the execution of control logic does not adversely affect application performance and resource usage. Third, in order to reduce the cost of development of adaptive behavior, control logic should be implemented in separable modules that can be modified, extended, and reused across different systems without affecting the main business logic of an application.

A flexible architecture is therefore essential to support the efficient implementation of adaptive behaviors in a non-intrusive way [5]. In this paper we present such an architecture framework, Adaptive Server Framework (ASF), to support the development of adaptive behavior for server side components running on application servers. The contributions of our approach, as embodied in ASF, are:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SEM'06 November 10, 2006, Portland, Oregon, USA.
Copyright 2004 ACM 1-58113-000-0/00/0004…$5.00.

1. ASF provides an extensible architecture framework for building *application-specific* adaptive behavior into server applications.
2. ASF supports a clear separation between application business logic and adaptive control logic.
3. ASF introduces low overheads in terms of performance and memory footprint.
4. ASF is built on standard interfaces and is portable across different application server platforms (at this stage, J2EE application servers only).

2 RELATED WORK

The blueprint for software architectures in autonomic computing is described in [6]. The challenges of developing autonomic computing systems are addressed at two levels, namely developing self-managing, self-tuning and self-adaptive autonomic elements and that of the interactions between autonomic elements. Reference architectures for autonomic computing have been developed from research prototypes, for example, Unity [3]. The explicit properties assumed by Unity are that each autonomic element must be self-managing internally and be self-healing locally. Based on the assumptions, Unity focuses on the interactions between autonomic elements. Interfaces are defined to restrict the communication between elements via web services, and elements register and locate services through a repository using policies specified for services. In contrast, in this paper, we focus on how to implement self-managing, autonomic behavior for a new or existing single server application.

The Rainbow project [4] proposes an architecture-based approach with the emphasis on adaptive strategies and techniques for detecting architectural styles at runtime. One common design principle of Rainbow and ASF is that adaptive control should be modular and separated from the managed application, interacting with the application in a non-intrusive way. However, in Rainbow, adaptation is predefined based on the architectural styles of the system and it doesn't address how customized adaptive behavior can be incorporated into an existing systems.

Research in reflective and adaptive middleware also provides infrastructures that can adapt their quality-of-service provision based on environmental needs (e.g.[12]). These technologies facilitate dynamic adaptation of a middleware platform by applications in ways that were not anticipated during its design [11]. Adaptation is driven by applications using an API that the underlying middleware platform supports. Such platforms are *translucent*, allowing applications access to components inside the middleware. This contrasts with our approach in ASF, which separates control of adaptive behavior from the application's business logic.

Other tools and run-time techniques to support the construction of adaptive applications are reported in [1]. Efforts have focused on designing autonomic services in application server technologies to make the deployed servers less costly and complex to manage. These solutions focus on system manageability and are not flexible enough to address application specific needs for functional adaptation.

Within an autonomic element, analytical models and intelligence play key roles in controlling and guiding adaptive behavior. [13] and [15] demonstrate how models for analysis can be useful in reconfiguring system resources subject to changing workloads. In this paper, models for analysis are deployed as ASF control components to control adaptation at the application level.

3 ADAPTIVE SERVER FRAMEWORK ARCHITECTURE

The key design principle of ASF is to separate the implementation of adaptive behavior from the server application business logic. Figure 1 depicts an overview of how ASF interacts with applications and the underlying application server platforms. ASF defines a component architecture for implementing dynamic, adaptive behaviors, such as managing the lifecycle of control components and coordinating their communications.

An adaptive implementation using ASF components runs in an adaptive engine. This engine interacts with the application server, monitors the runtime environment, analyzes collected data, and changes the application's behavior or the server's configuration to fulfill the business goals specified in policies. Unlike other architectures proposed [1][3][7][14], the implementation of ASF components for customized adaptation is developed and deployed externally to the applications, providing clear separation of concerns.

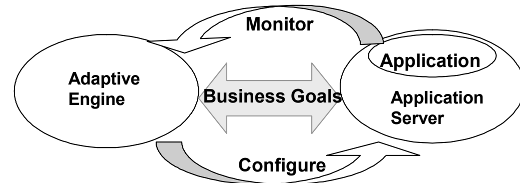


Figure 1 Anatomy of ASF

This is an especially attractive approach for converting 'non-adaptive' legacy applications into self-managed autonomic applications. For example, the business logic of a legacy application is typically complex, and may be purchased from third parties, which means modifying the business logic is not cost-effective or even possible. Using ASF, adaptive behavior can be plugged in to a server application transparently to the business logic, requiring no modifications.

In addition, ASF executes on top of the underlying middleware layer (in this paper J2EE), and hence does not require infrastructure modifications to the application server hosting the adaptive components. Each adaptation serves a specific goal for quality attributes, such as performance, scalability and security. More than one adaptation can be implemented separately and deployed into the application server. This makes the architecture extensible for dealing with the changing requirements for adaptive behavior over an application's lifetime.

3.1 Layered Architecture

The conceptual model of ASF is implemented in a layered architecture to manage control components and facilitate their interactions with the underlying application server platforms. As shown in Figure 2, the capabilities of ASF to support the non-intrusive adaptation are encapsulated in the *Management Layer* and *Adaptive Component Layer*.

Management Layer: our goal is to build adaptive server applications without requiring changes to or specialized services from the application server. This is achieved by introducing the Management Layer between the ASF engine and components and the hosting application server. The Management Layer includes utilities and mechanisms that can be used to monitor the runtime behavior of the application and the underlying platform, and change the configuration settings. The J2EE JMX (Java Management Extensions) based management architecture is leveraged in ASF for Java-based server applications [8]. Most J2EE application servers utilize JMX to both implement the internal server management and provide APIs for hosted applications to retrieve and set the state of the application server configuration [8].

Another advantage of implementing the Management Layer based on JMX is to achieve the portability of ASF across different application server architectures. Through JMX, the dependency on application server specific features is minimized to reduce the porting effort for ASF to different J2EE platforms.

Adaptive Component Layer: ASF executes on top of the Management Layer. It collects data on the application server's environment, tunes the server configuration and/or changes its behavior through the Management Layer interfaces. In this way we minimize the dependencies of ASF on the specific features of the underlying application server, meaning the same implementation can be deployed on different application servers.

The application server, the Management Layer and the Adaptive Component Layer together form the middleware platform that augments applications with adaptation. The application is not aware of the existence of the adaptive components and its development concentrates on the business logic implementation. This is especially useful for introducing adaptation into existing applications, of which access to the source code is not practical.

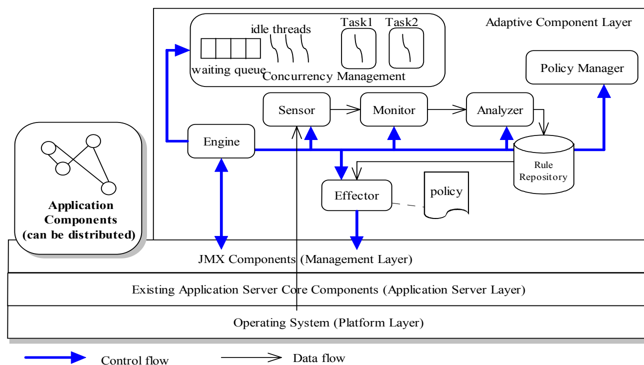


Figure 2 The layered ASF architecture

3.2 The Component Model of ASF

An adaptive application can be viewed as a set of interacting control components that collaborate to achieve the desired adaptive behavior. This requires a common set of underlying capabilities including monitoring an application's runtime environment, analyzing collected data, and changing the runtime configuration. An ASF component is the basic building block in the framework, from which specialized components, customized for an application, are derived. The basic components are:

Sensor: a component that provides probes to detect environmental measures, and collects data by sampling without analyzing them, such as the CPU usage, network connection speed, arrival rate of requests, and memory usage.

Monitor*:* a component that aggregates, correlates or filters sampling data collected by sensors. It composes a detailed view of the system's states or metrics for further analysis. The design of separating data collection and data filtering allows execution of these two tasks in parallel.

Analyzer: a component comprising one or more analytical models for use in implementing the required adaptive behavior. It takes monitored data and processes it to understand the current system state or predict possible future application states..

Repository: a component that stores the output of the decisions from the analyzer. These can be reused if the same application state is encountered, reducing the analysis overhead required.

Effector: a component that makes adaptive behavior occur. It changes the behavior of the server application directly by executing a different algorithm, or indirectly by modifying the configuration settings of the application server.

Policy manager: a component that manages policies by taking input conditions, parsing the policy descriptions and producing one resulting policy to take effect from several policies, based on the priority of each policy. The policy manager is also responsible for resolving conflicting polices. ASF currently only supports simple action-based policy reasoning [10].

QueuedExecutor: a component that hosts a thread pool and an associated wait queue. It provides a means of managing threads for implementing adaptive behavior. If the number of thread execution requests exceed the capacity of thread pool, requests will be either queued or rejected based on a defined handling policy.

Engine: a component that bootstraps all other components and acts as their manager. The engine together with other components it manages forms a deployment archive dedicated to one adaptation concern of the application under management.

Along with the basic component types defined in ASF, the component model also defines the interfaces associated with components, including:

Lifecycle interfaces: enable the engine to determine the state of an element and to cause the state to change, for example as enabled or disabled.

State management interfaces: permit a component to expose its attribute values.

Message-based communication interfaces: allow a message from a message producer component (e.g. a sensor) to be passed asynchronously to message consumer component (e.g. a monitor).

Concurrency interfaces: provide a way of decoupling task submission from the mechanics of how each task will run, including details of thread use and scheduling. It defines how asynchronous requests are processed concurrently by the QueuedExecutor.

Policy interfaces: are used by the policy manager to parse the XML policy description and query the policy information specified in it.

Bootstrap interfaces: provide the mechanisms for the adaptive engine bootstrap process to synchronize with the external services on which ASF depends.

3.3 Control Loop Component Composition

Individual ASF components are connected to form a control loop. A control loop provides the logic for adaptation. It includes at least a sensor, a monitor, an effector and an analyzer. It monitors the execution of the application, analyzing the data collected and taking actions according to policies defined by administrators/designers that govern adaptation.

Initially, a control loop is defined in a configuration file, which implicitly specifies the components involved. Each component has a set of attributes, which can be used to specify a variety of information needed to initialize and manage a component. For example, a CPU monitor has a frequency attribute, which specifies how often samples of CPU usage are collected. The control loop specification also determines how components interact and coordinate. For example, a component has attributes to specify the message it produces and the list of the components consuming this message (see Figure 3).

```
# Declare Sensors
SENSOR_LIST=BandwidthSensor,CPUSensor
# Specify BandwidthSensor's attributes
BandwidthSensor.CLASS=
au.com.nicta.sensor.impl.BandwidthSensor
BandwidthSensor.PRODUCED_MESSAGE_LIST=
ClientConnectionMsg
BandwidthSensor.ClientConnectionMsg.CONSUMER_LIST=
ImageScaleMonitor
BandwidthSensor.FREQUENCY=1
BandwidthSensor.MAX_FREQUENCY=50
BandwidthSensor.FREQUENCY_STEP=5
BandwidthSensor.MONITOR_INTERVAL=1000
```

Figure 3 Screenshot of the component configuration

This message-based communication pattern is illustrated in Figure 4. Message producers have the *dispatchMessage()* method that dispatches messages to the *QueuedExecutor*. The executor allocates a thread to iterate over all associated message consumer elements and invoke their *handleMessage()* callback method to process the message sent to it.

In ASF, a composition can be static or dynamic. A static composition is created when the engine starts. The engine initializes instances of each component specified in the configuration file, registers them in a naming service and binds their instances together. Static binding is necessary as the components involved in a control loop must be bound before they can communicate.

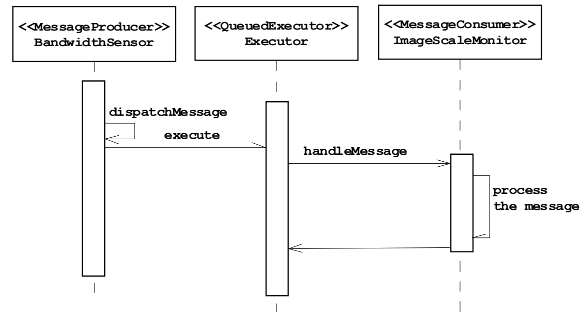


Figure 4 ASF component communication

Components can also be dynamically composed at runtime. The configuration file provides the engine with information on all the components it manages. At runtime a component instance can be created and invoked by its message-based communication interfaces through Java reflection. Dynamic composition is useful for the interactions and coordination between elements across different engines that must coordinate to achieve some adaptive behavior, for example in a distributed server cluster.

The implementation of a control loop is highly application specific. There can also be more than one control loop required to achieve the desired adaptation. A detailed example of devising multiple control loops is illustrated in section 5.2.

3.4 Policy Management

At runtime, the behavior of ASF components inside a control loop is determined by the state of the system under management based on the policies specified. A policy is a representation of desired behavior or constraints on behaviors defined in a standard external form [10]. A decision to take an action may depend on more than one policy. If a policy's conditions are met, an adaptation strategy is invoked based on that policy's definition. The evolving state of the application may lead to different policy decisions, which in turn may switch the control loop to different execution paths.

An XML schema is defined to describe the policy associated with an ASF component. Basically a policy consists of five elements: name, scope, condition, decision and priority. The scope element describes the specific scope that applies for this policy, and it is also used by the policy manager to categorize policies. As ASF currently only supports action based policy management, the condition element defines the condition expression with variables and operators such as greater, equal, not equal and less. The decision element is a description of the action and the actual implementation is fulfilled in ASF components. The priority (values from 1-10) is utilized by the policy manager to determine a policy to act on if a conflict of policies occurs.

In terms of the relationship with standard policy specifications, WS-Policy defines a container for assertions, and ASF policies can be assertions contained in a WS-Policy document. Figure 5 shows a simple policy example controlling the CPU utilization. If the CPU utilization is greater than 80%, then the action of 'TuneConfiguration' is to be taken, and this policy is of high priority with value 10.

```
<asfpl:Policy decisionName="TuneConfiguration" policyEnabled="true"
    policyName="CPUOverleadPolicy"
    xmlns:asfpl="http://www.nicta.com.au/amp/namespaces/policy"
    xmlns:exp="http://www.nicta.com.au/amp/namespaces/policy/expressions/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <asfpl:Description>CPU Policy Description</asfpl:Description>
    <asfpl:Condition>
        <exp:Greater>
                <exp:Property propertyName="CPUUtilization" />
                <exp:Constant>
                    <Value>0.8</Value>
                </exp:Constant>
        </exp:Greater>
    </asfpl:Condition>
    <asfpl:Decision name="TuneConfiguration">
        <asfpl:Action name="TuneConfiguration" />
    </asfpl:Decision>
    <asfpl:Priority>
        <Importance>10</Importance>
    </asfpl:Priority>
    <asfpl:Scope>
        <asfpl:ScopeType>
            <Value>cpuScope</Value>
        </asfpl:ScopeType>
    </asfpl:Scope>
</asfpl:Policy>
```

Figure 5 An example of policy description

4 CASE STUDY

When using ASF to develop customized adaptive behavior for server applications, the focus is on the control logic design. This involves the following tasks.

1. Derive a description of the adaptive behaviors that can meet the business goals.
2. Determine the control loops for implementing the adaptive behavior. For each control loop, the following tasks are carried out:
    - Identify parameters that characterize the adaptive behavior, along with their dependencies.
    - Devise analysis models that can represent the relationship between these parameters.
    - Determine the components involved in each control loop and how they interact. Normally, sensors and monitors are used to collect the values of measurable parameters, such as CPU usage and the arrival rate of requests. These values are fed into analysis models, which are embedded in analyzers. The output of the analyzers indicates how effectors perform adaptive behavior, such as changing the system's configuration.
3. Implement each component using ASF. The designer extends the basic components in ASF and implements their interfaces with the control logic needed to fulfill the business goals. Designers leverage the existing services, lifecycle management and concurrency control provided by ASF.
4. Deploy and execute the adaptive engine.

4.1 Adaptive Image Server

To illustrate the usage and validate the contributions of ASF in terms of performance, we have implemented an adaptive image server application. A simple high level depiction of this application is illustrated in Figure 6.

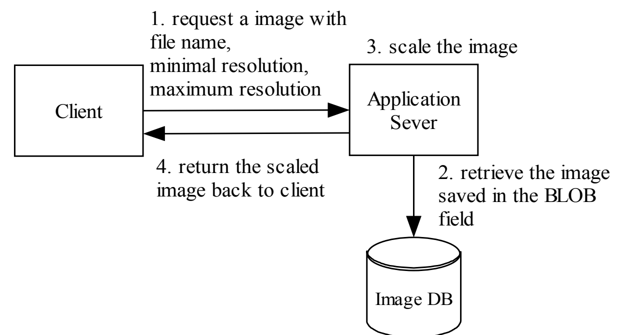


Figure 6 Image Retrieval Use Case

A client sends a request to the application server for a specified image along with a minimum and maximum resolution for the image. By default, the maximum resolution is 1.0, which means the original image is returned without scaling. The application server hosts the image processing application, which sends a request to retrieve the image from a database, where the image is stored as a BLOB (Binary Large Object). Without adaptation, the image server scales an image to the maximum resolution requested and returns it to the client.

4.2 Analysis

The business goals of the application are to improve the throughput and reduce the response time of the image server. Given that clients request a minimum and a maximum resolution for an image, the application is free to choose the resolution and image quality it delivers in order to optimize its performance. Scaling an image takes CPU time, and the image size affects image transport time. Hence the application can adaptively select the image resolution based on a model of the scaling computation cost and network latency.

It is not immediately obvious how image scaling with different level of resolution and quality affects performance. Hence the first design task is to find a relationship between the scaled image size, scaling time, and image quality. This was done by taking empirical measurements of scaling 100s of images with a size from less than 1KB to over 2MB.

The results demonstrate that the scaled image size depends on both resolution and quality, while the image scaling time is most affected by the resolution and the effect of quality on image scaling time is not significant. The higher the resolution or quality, the larger the final image size is. In addition, the image scaling takes longer as the resolution increases.

From these measurements, we can infer that image scaling time affects the application's response time, and the scaled image size determines the delay of transferring the images over the network for a given network speed.

Based on this preliminary analysis, the dependency between these characteristics is shown in Figure 7. In order to simply the analysis, here we do not consider the dependency between resolution and quality. We can derive a description of the adaptive behavior for this application as:

Based on the workload of the server and network connection speed, the server adaptively returns images at different levels of resolution and quality that can both meet the client requirements for the images and also optimize the performance of the application server under peak load.

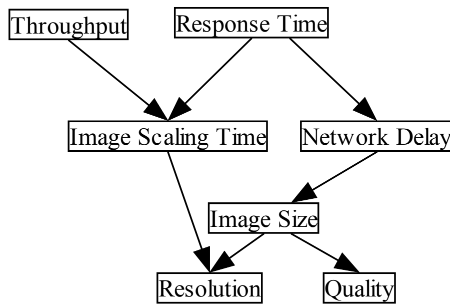


Figure 7 The metric dependencies

Designing the control components to fulfill this adaptive behavior requires an analysis model to represent the relationship between the CPU usage of the application server, the network speed, and the resolution and quality of the image to be scaled. Due to the space limitations, we omit the development of the analytical model.

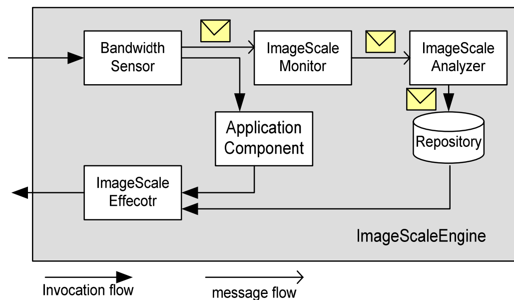


Figure 8 The control loop for scaling an image

The design of the adaptation trades off image resolution and quality against image processing time and size to reduce the transport overhead over network connections. Based on the above analysis, we can design and implement the following components to create a control loop to determine how an image is to be scaled, as shown in **Figure 8**.

- `ImageScaleEngine` is the engine component responsible for bootstrapping and managing all the other control components.
- `BandwidthSensor` intercepts a client request and detects the client connection network speed $B_{network}$ and request arrival rate λ. It forms a message with these details and sends it to the monitor. It also assigns a unique *id* to each invocation to differentiate invocations from different clients requesting the same image.
- `ImageScaleMonitor` takes the `BandwidthSensor` input message, attaches the CPU usage to the message and sends it to the analyzer. The CPU usage is collected from the CPU sensor, which is described below.
- `ImageScaleAnalyzer` implements the analytical model that represents the relationship between metrics in Figure 7.
- `ImageScaleEffector` retrieves the record from the repository. If there is no record stored, it just scales the image with the maximum resolution required by the client, otherwise it intercepts the return method of the application's invocation and replaces the return result with the image scaled according to the quality and resolution analyzed.

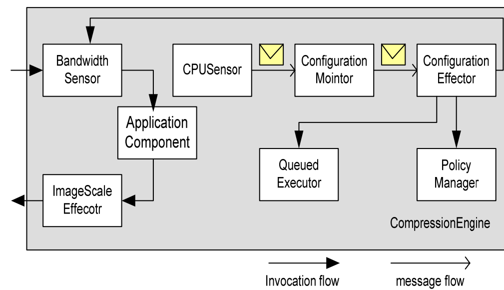


Figure 9 The control loop for CPU usage

The adaptive engine and the server application share the underlying platform resources (J2EE server, operating system, CPUs, memory). Consequently, the overall performance of the resulting application should be taken in to account to prevent CPU saturation and thus performance degradation. Therefore if the system is under heavy load, the concurrency level of adaptive behavior should be tuned to reduce the overhead caused by the adaptive engine. To achieve this, another control loop is used, as illustrated in **Figure 9**. Two action-based policies are defined for controlling the CPU usage. Simply, these are:

- Policy One: IF (CPU usage>= upper bound) THEN (tune the frequency and thread pool size)
- Policy Two: IF (CPUusage<lower bound) THEN (reset the original settings)

The *CPUSensor* detects the CPU usage for every interval. The default interval is one second. It sends the CPU usage samples as a message to the *ConfigurationMonitor*. If half of the samples during the last ten intervals exceed a configurable upper threshold of CPU usage, the monitor sends a tuning message to the *ConfigurationEffector*. Otherwise if half of the samples in the last

ten intervals are below the lower CPU usage threshold, it sends a reset message to the *ConfigurationEffector*[1]. The *ConfigurationEffector* is responsible for setting the frequency and thread pool size of the scaling engine elements.

We have developed a Queuing Network model to facilitate the tuning of the thread pool size and sampling frequency. The model captures the relationship between overall response time, the thread pool size, the sampling frequency and the arrival rate of requests and the CPU usage. Due to the space limitations, we omit the detailed descriptions of this model.

4.3 Case Study Implementation

We implemented the case study on JBoss Server 4.0.1 with JRE 1.5 using ASF. The adaptive image processing engine extends JBoss' MBean interface and forms a *.sar deployment achieve. JBoss's deployment management components automatically initialize an instance of the engine and register it with the JBoss MBean server.

The *BandwidthSensor* and *CPUSensor* detect network speed and CPU usage respectively. They invoke native code libraries using Java Native Interface (JNI) calls on Windows XP. The network speed library implements the 'Ping' command using ICMP (Internet Control Message Protocol). The CPU usage library accesses to Windows XP performance counters through the Win32 API. The *BandwidthSensor* and *ImageScaleEffector* are deployed as JBoss interceptors [8], so that the signature of an invocation request and its return value can be captured.

We also developed a timer utility with resolution of 1 millisecond using JNI and the Win32 API. We use this timer to instrument the relevant operations at the beginning and at the end of the code.

4.4 The Test Bed Setup

A Java client application simulates the workload by starting a number of threads and simultaneously sending requests to the server. Each client randomly picks a file name as its parameter for the request to the server, and the server component (a session EJB) returns the corresponding image as a byte stream. Our test data has images stored in an Oracle 9i database as BLOBs, with sizes from 800 bytes to 2.3MB. Our lab environment support 100Mbps Ethernet, with the throughput of a network connection approximately 960Kbps. The client, application server and database machines are identical. All are workstations with Dual Intel Xeon 3.00GHz CPUs and with 3G RAM, running Windows XP.

5 EVALUATION

We evaluate the case study application using two methods. The first measures the overhead of the image scaling engine in term of its performance and memory footprint. The second compares the response time with and without the adaptive engine solution.

By inserting timing probes in to components that consume CPU in the control loop and summing the service time, the overhead of adaptation, excluding the time spent on scaling the image is measured at approximately 5ms per request. The overall size of Java classes compiled for this adaptive application is 297k. We also measured the runtime memory footprint as approximately 3.65MB.

[1] Only the default settings that have been changed are reset.

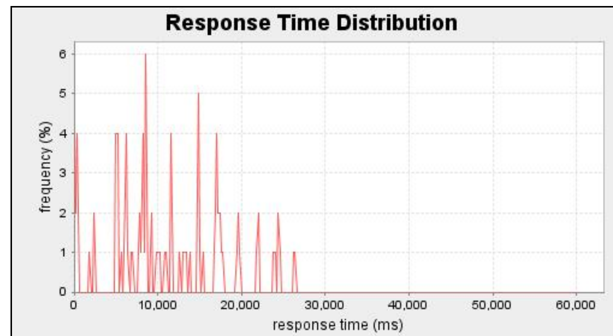
We first compared the throughput of the JBoss server with and without adaptive behaviour given a fixed number of requests of 10 clients. We categorized the workload according to the size of the image into four groups, namely less than 10KB, 10KB~100KB, 100KB~500KB, and 500KB~2MB. The client requests for images are evenly distributed among the four groups. The results are listed in Table 1.

We then generated a workload with varying arrival rates by periodically changing the number of emulated clients. The number of concurrent clients was increased in steps of ten every two minutes from a starting number of five until a maximum of 65 concurrent clients are sending requests to the server. After that, at two minute intervals, the number of clients is decreased by 10 until 35 remain. The intent of this workload is to mimic sustained bursts of increasing workload against a backdrop of moderate activity. Each step in the workload produces a different plateau of workload level. Therefore the workload stimulates the engine to adapt to CPU usage on and below the peak level.

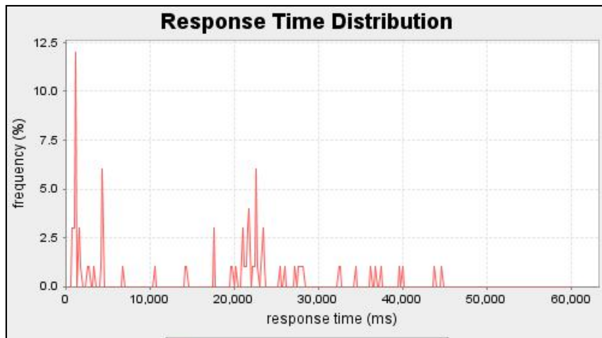
Image Size	Non-adaptive TPS	Adaptive TPS	% Improvement
Small (10KB)	54.25	89.23	64.48%
Small Medium (10~100KB)	5.78	8.95	54.84%
Medium (100~500KB)	1.94	3.10	59.79%
Large (500KB~2MB)	0.25	0.31	24%

Table 1 Comparison of throughput under fixed workload

The entire test lasts for twenty minutes. The same workload is applied to the server application executing with and without adaptation. The JBoss server configuration is exactly the same for both tests. Figure 10 shows the client response time distribution for these two tests. With adaptation executing, 90% of response times are within 23 seconds and the overall throughput is 240 transactions per minute. Without adaptation, 90% of response times are within 36 seconds and the overall throughput is 187 transactions per minute. This experimental evaluation verifies that our framework is a lightweight implementation of an architecture that can improve the performance of server applications through adaptive behavior.



(a) adaptive behavior enabled



(b) adaptive behavior disabled

Figure 10 Comparison of response time distribution

6 CONCLUSION

In this paper we present an architecture framework known as ASF for developing adaptive behavior for applications running on J2EE application servers. The framework provides the basic components and services to support designers in building customized adaptive behavior for their application's requirements. The layered ASF architecture exploits standard management services, to insulate the adaptive logic implementation from the underlying application server without changing the code of either the application or the application server. This reduces the dependency on application server specific features, and thus the framework is extensible and portable.

We demonstrate the use of ASF in building an adaptive image server application. The performance evaluation of the application clearly illustrates that the overheads introduced by ASF are low, and that performance benefits can be gained by using adaptive engines built using the framework.

Our experience in building adaptive behavior for the image server application has provided some insights into developing adaptive server applications:

- The adaptive logic design and implementation are driven by the application's requirement for adaptation. The high-level requirements need to be translated into polices that can be evaluated to produce decisions. Analytical models play a critical role in reasoning over the system's behavior and providing the necessary rules for adaptation.
- Developing adaptive engines is difficult. Designers need a programming model such as ASF to solve many of the low level problems of interacting with the application and performing efficient analysis and adaptation. We believe that developing adaptive engines is a specialized task, and hence solutions should shield application programmers from the inherent complexity.

This work only focuses on single node server applications. It is essential to extend this work towards fully distributed application environments. We are currently extending this framework with support for web service resource management. The coordination of elements across different nodes and policy management regimes in a distributed environment present significant challenges for our future research.

7 REFERENCES

[1] Abdellatif, T., Enhancing the Management of a J2EE Application Server using a Component-Based Architecture. Proc. of the 31st IEEE/Euromicro Conf., (2005).

[2] Bigus, J. P., Schlosnagle, D. A., Pilgrim, J. R., Mills III, W. N. and Diao, Y.: ABLE: A toolkit for building multiagent autonomic systems. IBM Systems Journal, 41(3), (2002), 350-371.

[3] Chess, D. M., Segal, A., Whalley, I., White, S. R., Unity: Experiences with a prototype autonomic computing system, ICAC'04: Proc. of 1st Intl. Conf. on Autonomic Computing, (2004), 140-147.

[4] Garlan, D., Cheng, S., Huang, A., Schemerl, B., Steenkiste, P., Rainbow: Architecture-based self –adaptation with reusable infrastructure, IEEE Computer, vol. 37, no. 10, (2004), 46-54.

[5] Gorton, I., Liu, A., Brebner, P., Rigorous Evaluation Of COTS Middleware Technology, IEEE Computer, vol. 36, no.3, (2003), 50-55.

[6] IBM Autonomic Computing, An architectural blueprint for autonomic computing, white paper, (2004). http://www-03.ibm.com/autonomic/pdfs/ACBP2_2004-10-04.pdf

[7] Jordan, M., Czajkowski, G., Kouklinski, K., Kirill Skinner, G.: Extending a J2EE™ Server with Dynamic and Flexible Resource Management, Middleware (2004).

[8] Java Management Extensions (JMX) Specification, http://java.sun.com/products/JavaManagement/JMXperience.html

[9] Kephart, J. O.: Research challenges of autonomic computing, ICSE '05: Proc. of the 27th Intl. Conf. on software engineering, (2005), 15-22.

[10] Kephart, J. O., Walsh, W.: An artificial intelligence perspective on autonomic computing polices, IEEE 5th Intl. Workshop on Policies for Distributed Systems and Networks, (2004) 3-12.

[11] Kon, F., Costa, F., Campbell,R., Blair, G.: The Case for Reflective Middleware. Communications of the ACM. Vol. 45, No. 6, (2002), 33-38.

[12] Kon, F., Román, M., Liu, P., Mao, J., Yamane, T., Magalhães, L. C., Campbell, R.: Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. IFIP/ACM Intl. Conf. on Distributed Systems Platforms and Open Distributed Processing, (2000).

[13] Menascé, D.A., Bennani, M. N., and Ruan, H., On the Use of Online Analytic Performance Models in Self-Managing and Self-Organizing Computer Systems, LNCS vol. 3460, (2005).

[14] Parashar, M., Liu, H., Li, Z., Matossian, V., Schmidt, C., Zhang, G. and Hariri, S., AutoMate: Enabling Autonomic Grid Applications, Cluster Computing: The Journal of Networks, Software Tools, and Applications, Vol. 9, No. 1, (2006).

[15] Wildstrom, J., Stone, P., Witchel, E., Monney, R.J., Dahlin, M., Improve performance with self-configuring distributed systems, IBM develperWorks, (2005). http://www128.ibm.com/developerworks/autonomic/library/ac-selfhw/