An Extensible and Lightweight Architecture for Adaptive Server Applications

Ian Gorton, Yan Liu, Nihar Trivedi

1Pacific Northwest National Laboratory, U.S.A
2National ICT Australia Ltd.*
3School of Information Technologies, University of Sydney
ian.gorton@pnl.gov, {jenny.liu,nihar.trivedi}@nicta.com.au

ABSTRACT

Server applications augmented with behavioural adaptation logic can react to environmental changes, creating self-managing server applications with improved quality of service at runtime. However, developing adaptive server applications is challenging due to the complexity of the underlying server technologies and highly dynamic application environments. This paper presents an architecture framework, the Adaptive Server Framework (ASF) to facilitate the development of adaptive behaviour for legacy server applications. ASF provides a clear separation between the implementation of adaptive behavior and the business logic of the server application. This means a server application can be 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 behavior and the low overhead introduced by ASF.

KEY WORDS: Adaptive, Server Applications, Autonomic Computing, Components, Performance

*National ICT Australia is funded through the Australian Government's Backing Australia's Ability initiative, in part through the Australian Research Council.
1. INTRODUCTION

Application server platforms encompass advanced middleware technologies and provide a standardized service layer to support the development and deployment of distributed, server-side applications. Application server technologies such as CORBA, J2EE and .NET can be used to build server applications that support various levels of quality-of-service, making them suitable platforms for software systems with demands for high performance and reliability.

However, these mission-critical server applications remain challenging to construct and maintain. With the rapid evolution of technologies, fully fledged application servers are becoming more and more complex application hosting environments, with hundreds of parameters to be tuned, configured or indirectly stimulated or triggered when components of applications are deployed and executed. Moreover, 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 statically optimize the quality goals (such as performance and reliability) of an application in all circumstances.

To reduce application configuration and management complexity, creating server applications with self-management and adaptation capabilities is being explored. Adaptation in application servers can be achieved using three different strategies, namely structure adaptation, architecture adaptation and behaviour adaptation. Structure adaptation means changing the type of application components, such as a method signature. Aspect-oriented programming techniques have been used to achieve structure adaptation [5]. Architecture adaptation uses a system’s software architectural model to
monitor and reason about the system. This results in the changes in the structure of components and their interactions [7]. For example, a new server instance is launched within a server cluster when the workload increases. Behavioural adaptation, as described in this paper, focuses on changing the execution of existing components in a non-intrusive way, such as changing the configuration of a component, and intercepting the requests and replies of a method invocation. These three techniques can also be combined to deliver highly adaptive solutions.

Initial steps towards adaptive technologies are following two main paths. One is building self-management capabilities into application server platforms and creating self-managing solutions through prepacked components and services. For example, IBM’s WebSphere application server product is currently integrated with other IBM autonomic computing toolkits such as load balancing and log analysis [12].

The second path is exploring enhancing existing application components with adaptation using ad-hoc programming language features such as complex conditional expressions, reflection and exceptions. However these approaches have significant drawbacks, as the adaptation and application business logic are interweaved and tightly coupled, leading to complex testing and poor extensibility and maintainability. Moreover, for many legacy applications such as off-the-shelf packages, modifying the source code is often not possible or desirable.

A solution to these problems is to separate the adaptation logic from the application business logic. This requires software infrastructure support so that the application programmer is relieved of concerns regarding the adaptive logic implementation. For server applications executing on application server platforms, this approach can be
achieved by extending existing application server platforms with adaptive mechanisms [7]. By augmenting existing middleware, it becomes possible to transparently build adaptive capabilities into existing applications. It also promotes an attractive adoption path for new adaptive applications that remain based on standard application server platforms.

Other approaches for developing adaptive server applications have been investigated [2] [23]. However, 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.

Regardless of the precise approach adopted, 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, as adaptation must occur at runtime in response to changes in the application’s environment. Second, the 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 needs to 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 behavior in a non-intrusive way [8]. In this paper we present such an architecture framework, the Adaptive Server Framework (ASF), to support the
development of behavioural adaptation for server side components running on application servers. The major contributions of our approach, as embodied in ASF, are:

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 J2EE application server platforms.
5. ASF demonstrates that adaptive behavior can be successfully built on top of existing middleware layers to facilitate the transparent augmentation of legacy applications with adaptive behavior.

The remainder of this paper describes ASF and a case study that we have implemented to demonstrate its suitability for building adaptive server applications.

2. RELATED WORK

Behavioural adaptation of server applications is encompassed within the autonomic computing research domain. The blueprint for software architectures in autonomic computing is described in [10]. 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 [4].
The explicit properties assumed by Unity are that each autonomic element must be self-managing internally and be self-healing locally. Based on these 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 consider a deployable server application to be an autonomic computing element, and we focus on how to implement self-managing, autonomic behavior for new or legacy single server applications.

The Rainbow project [7] proposes an architecture adaptation approach with the emphasis on adaptive strategies and techniques for detecting architectural styles at runtime. In Rainbow, adaptation is predefined based on the architectural styles of the system. An architectural model which is represented as a graph of interacting computational elements is used to reason about what architectural changes are needed. In comparison, ASF is more focused on a software framework supporting behavioral adaptation. Specifically it addresses how application-specific adaptive behavior can be incorporated into existing systems. 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.

Research in reflective and adaptive middleware also provides infrastructures that can adapt their quality-of-service provision based on environmental needs (e.g. [6], [9], [19]). These technologies facilitate dynamic adaptation of a middleware platform by applications in ways that were not anticipated during its design [18]. Adaptation is driven by applications using a reflective API that the underlying middleware platform supports.
Such platforms are consequently translucent, allowing applications access to components inside the middleware. This contrasts with our approach in ASF, which completely 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. [21] and [26] 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

The fundamental design principle of ASF is to separate the implementation of adaptive behavior from the server application business logic. This means the adaptation should be encapsulated into components external to the application implementation. Hence the goal of ASF is to provide infrastructure components and services to facilitate the construction of behavioral adaptation. This is achieved by creating a component model for developing adaptive control components, and providing associated services that support the deployment of adaptive components with J2EE application servers.

Figure 1 depicts an overview of how ASF interacts with applications and the underlying application server platform. ASF defines a component architecture for
implementing dynamic, adaptive control, such as managing the lifecycle of control components and coordinating their communications. Adaptive logic implemented using ASF components runs in an adaptive engine. ASF components interact with the application server, monitor the runtime environment, analyze collected data, and change the application’s behavior by adapting the response or setting the server’s configuration to fulfill the business goals. In this way, ASF helps developers to focus on the design and implementation of adaptive logic and insulates them from the complexities of the underlying J2EE platform.

3.1 Layered Architecture

To fulfill the goals of ASF, its various components and services are designed as reusable from application to application. We divide the ASF infrastructure into three layers to address the reusability of its components and services. ASF executes above the middleware layer, and the current implementation runs on top of J2EE, as shown in Figure 2.
**Adaptive component layer:** This layer supports a component model that defines the control components and interactions needed to implement adaptive behaviors. The control components collect data on the application server’s environment, tune the server configuration and/or change its behavior through management layer interfaces.

**Service layer:** ASF acts as a *container* of control components for adaptive behaviors. In order to manage these components, ASF provides a default implementation of useful services, such as lifecycle management, concurrent execution of ASF components, security management, and so on. The service layer provides infrastructure support for reusable ASF components services. For example, policies are application-defined, but the mechanisms for checking and enforcing polices provided by the policy management subsystem are generic, and therefore they can be reused as infrastructure services. By separating these general services into a specific layer, the tasks of developing adaptive behavior using ASF control components can be further simplified.

**Management layer:** ASF control components are executed on top of the management layer. The management layer includes utilities and mechanisms that can be used to monitor the runtime behavior of the application and the underlying platform, and reconfigure the application settings. Our goal is to build adaptive server applications without requiring changes to or specialized services from the underlying middleware. This is achieved by utilizing standard-based management protocols to collect status information and reconfigure an application server. For example, the J2EE JMX (Java Management Extensions) based management architecture is leveraged in our J2EE implementation of ASF [14]. Most J2EE application servers utilize JMX to both implement internal server management and provide APIs for hosted applications to
retrieve and set the state of the application server configuration [14]. Later in section 5.5, we provide an example of how ASF is deployed and interacts with the JBoss application server through JMX.

The adaptive component layer, the service layer and the management layer together form the platform that augments applications with adaptation. The application is not aware of the existence of the adaptive components, thus achieving our aim of separation of concerns.

![Figure 2 The layered ASF architecture](image)

### 3.2 Reuse in ASF

Adding adaptation to a server application utilizing the functionalities provided by ASF requires applying application-specific adaptation logic to customize the ASF infrastructure. The adaptation logic encompasses models that define parameters and their
relationship, behavioral constraints, policies and adaptation strategies. This adaptation logic is encapsulated in the implementation of ASF components.

For example, automatically configuring a server thread pool size based on the application workload requires an ASF component which monitors the application’s request arrival rate. This communicates with an analytical model representing the relationship between request arrival rate and the size of the thread pool implemented in a ASF analysis component.

Although the adaptive logic implemented by specific ASF component instances is application-specific, the common structure and behavior of ASF components required in different applications, including the types of components and their interaction patterns, can be abstracted into a component framework similar to EJB and CCM. For example, a monitor component may be implemented differently to collect CPU status, or network speed or the runtime server thread pool size. Regardless, the lifecycle methods are the same for different monitor instances.

Therefore the ASF component framework defines the types and interfaces of ASF components, as well as their communication patterns, and the framework is reusable from application to application. Details of ASF components are described in section 4. We will illustrate how ASF components are utilized for developing application specific adaptation in section 5 using a case study.

4. THE ASF COMPONENT MODEL

The kernel of ASF is the component model and its default implementation. The component model defines the basic types of control components, their interfaces, and
communication patterns. The common behaviors of each type of control component are abstracted as the component interfaces. ASF provides a default implementation of components to reduce the overhead of developing them from scratch. Also the component model leverages the ASF services, which further eases the development of adaptive behaviors.

4.1 Basic ASF Components and Interfaces

The key concepts in ASF are the components used for implementing the adaptive behaviors, for example, 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 default implementation of ASF provides component skeletons and glue for their communication. It also provides default component implementations for common adaptive behavior, such as detecting network speed, monitoring CPU usage and reconfiguration of platform settings. Figure 3 shows an example of several ASF components that form a control loop using message based communication. Message-based communications ensures that ASF components are loosely coupled and can be bound flexibly to form a control loop.

The framework also supports management of the number of threads used for executing ASF components. This is achieved by wrapping the execution body into a task, and ASF components utilize a thread pool that executes tasks on their behalf. The thread pool allows the designer explicit control over concurrency in executing the adaptive component tasks.
The framework provides the following major component types:

**Sensor:** a component that provides probes to detect environmental measures, and collects data such as the CPU usage, network connection speed, arrival rate of requests, and memory usage. Sensors only collect raw data and pass this on for further processing. Each adaptive application defines the set of sensors it requires.

**Monitor:** a component that aggregates, correlates or filters sampled data collected by sensors. It composes a detailed view of the system’s states or metrics for further analysis. This design, which separates data collection and data filtering, supports executing 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. An
example is using a queuing network model to analyze or predict the system’s throughput or resource usage.

**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. It can directly accept a monitor’s instructions to effect a decision or retrieve a decision record from the repository if the current state and resulting decision has already been encountered and stored.

**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. ASF currently only supports simple action-based policy reasoning [17].

**QueuedExecutor:** a component that hosts a thread pool and an associated wait queue. An execution task for a component is dispatched to the executor. It provides a means of managing the threads used in 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 form a deployment archive dedicated to one or multiple adaptation concerns for a specific application under
management. An engine also sets the management boundary for ASF components. Communication between ASF components deployed in different archives is achieved through the interfaces of their enclosing engine.

The ASF component model also defines the interfaces and bindings of components, as described below:

Lifecycle interface: Each ASF component also supports the Element interface (see Figure 4) that enables the engine to determine and change the state of an element (i.e. ASF component).

```java
public interface Element extends Serializable {
    public String getType();
    public String getName();
    public ElementInfo getInfo();
    public void setInfo(ElementInfo info);
    public void reloadInfo() throws Exception;
    public void enable();
    public void disable();
    public boolean isEnabled();
}
```

Figure 4 The Element interface of ASF components

State management interfaces permit an element to expose its attribute values. Each element is associated with an information object representing its attributes, for example the sampling frequency of a sensor.

Message-based communication interfaces allow a message from a message producer element (e.g. a sensor) to be passed asynchronously to message consumer elements (e.g. a monitor). This message-based communication pattern is illustrated in Figure 5. 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.
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. The concurrency interfaces are implemented using the JDK 1.5 concurrency package. These allow synchronous invocation through the `execute()` method or asynchronous invocation through the `submit()` method. The default implementation for the asynchronous interface also obtains the return result of asynchronous calls by applying the Futures design pattern [20]. ASF provides the default management of the concurrent execution of threads. Basically a number of threads are initially pooled within ASF as specified in the ASF engine configuration file. Depending on the contention incurred by concurrent execution of threads, ASF dynamically shrinks or expands the pool size. More details are described in section 4.3.

The interface for an executor is shown in Figure 6

```java
public interface AutonomicExecutor extends Element, Manageable {
    public Future submit(Runnable task);
    public Future submit(Callable task);
    public void execute(Runnable task);
}
```

Figure 6 Executor Interface
The body of the thread execution is wrapped in a Task class which implements the Java Runnable interface with no return results or Callable interface with return objects. Figure 5 shows the interactions between ASF message producer and message consumer components through concurrent interfaces.

Policy interfaces are the interaction mechanism for the policy manager and the policy actuator interactions. Basically the policy manager utilizes a PolicyUtility component to parse an XML policy description and query the policy information specified. It categorizes policies in terms of the scope that a policy can have an effect on. A scope is a string that describes particular resources or components that the policy affects. For example, the policy scope CPU indicates that the policy is concerned with CPU usage. (An example policy specification is depicted in Figure 9) An ASF component can be attached to one or more policies, and such a component becomes a PolicyActuator. A policy actuator has interfaces to attach/detach policies and enable/disable policies, and is responsible for enacting a decision for its enabled policies. The actual decision is determined by the policy manager based on the enabled policies. The policy manager is responsible for detecting conflicting policies, determining which policy should be carried out and returning this policy’s decision to the policy actuator.

Bootstrap interfaces provide the mechanisms for the adaptive engine bootstrap process to synchronize with the external services on which ASF depends. For example, an engine must wait for an authentication service to start before it performs other bootstrap operations. A callback method handleNotification() of the ASF adaptive engine is invoked once dependent services complete.
4.2 Control Loops and Component Bindings

Individual ASF components are connected to form a control loop. A control loop 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 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 messages it produces and the list of the components consuming this message (see Figure 8).

---

**# Declare Sensors**

```
SENSOR_LIST=BandwidthSensor,CPUSensor
```

**# Specify BandwidthSensor’s attributes**

```
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 8 ASF Component configuration**

Component bindings enable a component to request a service from another component or provide a service to others. In ASF, a binding can be static or dynamic. A static binding 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.

Components can also be dynamically bound at runtime. An engine has a list of all components it manages. A component’s information can be navigated by its component type through this list and a reference obtained using JNDI. Dynamic binding is useful for interactions between components across different engines that must coordinate to achieve some adaptive behavior. For example in a distributed server cluster, ASF components and their associated engines are deployed on each member of the server cluster. ASF
components across the boundary of a single engine need access to the JNDI of individual server maintained by a cluster to obtain the reference to another engine’s component.

At runtime, the behavior of ASF components 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 [17]. 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.

4.3 Concurrency Management

One of the goals of ASF is to minimize the performance overhead incurred in executing adaptive behavior. The control components that provide the adaptive behavior must execute in as short a time as possible so that they do not severely delay the request processing by the application business logic.

For example, sensor and effector components are inserted in the server request handling call chains used by the underlying application server\(^1\) [14]. They are therefore on the critical path to determine the response time of the server application. If sensors/effectors are synchronized with other components and must wait for responses to requests, they will degrade the overall performance significantly.

One solution is to increase the concurrency level of ASF by adding more threads. However there is a trade-off between concurrency and contention introduced by multi-

\(^1\) For example, customized request and response handlers can be deployed into application servers that intercept requests and responses, such as Axis handlers and JBoss interceptors.
threading. This problem is addressed in ASF by dispatching tasks asynchronously to the QueuedExecutor described in section 4.1 and 4.2 in order to control the concurrency level and minimize delays.

The implementation of QueuedExecutor provides improved performance when executing large numbers of asynchronous tasks due to the reduced per-task invocation overhead of creating threads. The QueuedExecutor thread pool and its wait queue also smooth out any increasing demand from threads caused by peaks in workload in the engine by controlling the size of thread pool. The thread pool size is an important parameter for tuning the performance of the adaptive engine as it directly affects the concurrency level.

The ASF architecture, with a dedicated QueuedExecutor component, represents a clear separation of concerns in managing the multithreading of adaptive behavior. The execution tasks are dispatched to the executor which is independent of the application server’s thread management. This provides the advantage of managing the concurrency of the adaptive engine separately. Tuning the thread pool size for the engine can be done without interfering with the application server’s settings.

### 4.4 Policy Specification

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 of this policy applies, 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 term of the relationship with standard policy specification, WS-Policy defines a container for assertions, and ASF policies can be assertions contained in a WS-Policy document. Figure 9 shows a simple policy example controlling the CPU utilization, that is 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.

```xml
<ASFp1:Policy decisionName="TuneConfiguration" policyEnabled="true"
        policyName="CPUOverloadPolicy"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <ASFp1:Description>CPU Policy Description</ASFp1:Description>
  <ASFp1:Decision>
    <ASFp1:Condition>
      <ASFp1:Greater>
        <ASFp1:Property propertyName="CPUUtilization" />
        <ASFp1:Constant>
          <Value>0.8</Value>
        </ASFp1:Constant>
      </ASFp1:Greater>
    </ASFp1:Condition>
  </ASFp1:Decision>
  <ASFp1:Action name="TuneConfiguration" />
</ASFp1:Policy>
```

**Figure 9 Example of policy description**
4.5 Deployment of ASF Components

ASF components need to be instantiated for a specific application and deployed into the server that the application runs on. ASF utilizes JMX to implement the probe for J2EE application servers.

There are two ways of implementation this depending on the architecture of the application server. One is to deploy the ASF components, such as sensors and effectors as Interceptors through which any invocation to the J2EE server passes. This mechanism doesn’t require any changes in the client application. The other way is to wrap ASF components as JMX MBeans and deploy the MBeans into the MBean server supported by the J2EE application server environment. In this case, client applications need to delegate their invocations to MBeans, which redirect calls to application components. In this case, AOP techniques can be applied to wrap the MBeans specific code for an application server, which is then weaved in to the client wherever delegation to MBeans is required. In this way, instrumentation and interception can be implemented in MBeans without changing the implementation of application components. We can also separate the platform specific porting code from the client application implementation.

5. 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 needed to meet the business goals.
2. Determine the control loops for implementing the adaptive behavior. For each control loop, the following tasks are carried out:
3. Identify parameters that characterize the adaptive behavior in this control loop, along with their dependencies.

4. Devise analysis models that can represent the relationship between these parameters.

5. Determine the components involved in each control loop and how they interact. Sensor and monitor components 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 analyzer components. The output of the analyzers indicates how effector components perform adaptive behavior, such as changing the system’s configuration.

6. 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.

7. Deploy and execute the adaptive engine.

8. Test and evaluate the resulting application behavior.

Tasks 2-5 are iterative until the adaptive behavior satisfies the business goals. The process described above is a general guide for developing adaptive behavior for server applications. In the rest of this section we use a case study to specifically demonstrate how to use ASF to develop adaptive behaviors.

5.1 Adaptive Image Server

We have been building a case study to validate the design of ASF, based on an adaptive image server. A simple functional description of the image server application is
illustrated in Figure 10. 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, which means the original image is returned without scaling. The application server, which can be clustered, 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. With adaptation, the goal is to scale the image resolution so that the overall performance of the application can be optimized.

![Figure 10 Image Retrieval Use Case](image)

**Figure 10 Image Retrieval Use Case**

### 5.2 Problem 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 (see samples in Appendix) 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.

![Figure 11 The metric dependencies](image)

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 11. 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.

Designing the control components to fulfil this adaptive behavior requires an analysis model to represent the relationship between the throughput, the response time, the image scaling time, the network speed, and the resolution and quality of the image to be scaled. As our aim in this case study is to validate the ASF framework, we do not strive to design and implement an optimal model that can precisely capture the relationship for all these variables.

Instead, we use an approximation to simplify the modeling. This takes advantage of the fact that the throughput and response time depend directly on the image scaling time, which in turn depends on the CPU for computation. Therefore, when the CPU is approaching saturation, the resolution of the image can be sacrificed, using a lower resolution, so that the demand on CPU for image scaling is reduced. This will improve response times in periods of high request loads.

Based on this insight, we make an assumption that resolution is a function of the CPU usage of the application server, such that:

- When CPU usage increases the resolution degrades.
- When the workload is heavy and CPU usage is saturated, we assume the resolution returned is the minimum requested.
- When the workload is light and CPU usage is low, the resolution can be the maximum requested.

An exponential function fits well with our assumption. Let $Y_r (0 \leq Y_r \leq 1)$ be the resolution and $X_{cpu} (0 \leq X_{cpu} \leq 1)$ be the CPU usage. We can have a function as follows:
\[ Y_r = e^{-aX_{cpu}} \quad (a > 0) \]  

We can solve the value of coefficient \( a \) by assuming, for example, when the CPU reaches 100% utilization, the resolution is 50% of the original size, giving \( a \) as approximately 0.6931. Figure 12 shows this simple model for representing the relationship between CPU usage and resolution.

Note that the utility function derived in Equation (1) is simple and it satisfies our assumption listed above. Our major aim in this case study is to have a function that adequately represents the relationship between CPU usage and image resolution, rather than create an exact and complicated analytical model. More complex analytical models could be used, and it remains an aim for our future work to improve this simple model and investigate the impact of various analytical models on the efficiency of the adaptive behavior.

\[ Y = Y_{min} \quad \text{if} \quad Y_r < Y_{min} \quad \text{or} \quad Y = Y_{max} \quad \text{if} \quad Y_r > Y_{max} \]  

**Figure 12 A simple model for CPU usage and resolution**

In order to satisfy a client’s resolution requirement, the actual resolution \( Y \) used is determined as

\[ Y = Y_{min} \quad \text{if} \quad Y_r < Y_{min} \quad \text{or} \quad Y = Y_{max} \quad \text{if} \quad Y_r > Y_{max} \]  

28
Where \( Y_{\text{min}} \) and \( Y_{\text{max}} \) represents minimum and maximum resolution specified by the client requests.

After we determine the resolution of an image, we can use a similar approach to model the relationship between quality \( (Y_q, 0 \leq Y_q \leq 1) \) and network speed \( (X_{\text{net}}) \) in Equation (3). The higher the network connection speed, the higher quality image can be delivered.

\[
Y_q = e^{bX_{\text{net}}} + k \quad (b > 0) \tag{3}
\]

According to the specification of the JDK AWT image scaling API, quality values of 0.75, 0.5 and 0.25 roughly mean high, middle and low quality respectively. We assume if the network is slow then the quality is set to the low value (0.25), while for fast network speed such as 1Mbps the quality is set to the high value (0.75). We can solve the coefficients \( b \) and \( k \) and have a simple function shown in Figure 13.

![Figure 13 A model for network speed and quality](image)

5.3 Determining the frequency and concurrency levels

The computing cycles used by the image processing engine and the application server are major sources of CPU cycle consumption. They can be modeled in a simple open queueing network model with two load independent multiple servers, one for the image
processing engine and one for the application server. In Figure 14 \(m\) and \(n\) represent the thread pool size of each respectively, \(\lambda\) and \(\lambda_{img}\) represent the request arrival rate at the application server and the image processing engine respectively. \(\lambda_{img}\) depends on the sampling frequency \(f\), for example \(f=1\) means doing adaptation for every client request, and \(f=3\) means sampling every third client request.

\[
\lambda_{img} = \frac{\lambda}{f} \quad (4)
\]

Applying the MVA algorithm for an open queueing network model [21], we can derive the CPU utilization and response time of the compression engine below

\[
U_{img} = \lambda_{img} (D_{img} + D_{adp}) \quad (5)
\]

\[
R_{img} = \frac{(D_{img} + D_{adp})/m}{1 - \lambda_{img} \times (D_{img} + D_{adp})/m} + \frac{(m-1)(D_{img} + D_{adp})}{m} \quad (6)
\]

From Eq.(4)(5), it is clear that to reduce the CPU usage when it exceeds the specified upper bound threshold, such as \(U_{img} \geq upper\_bound\), we can increase the frequency value to reduce the arrival rate \(\lambda_{img}\) in the adaptation engine. Note that simply put \(upper\_bound\), as 100% is definitely not the optimal setting. When the CPU is already saturated, there is no computing resource available for the reconfiguration operations. In fact, as the CPU usage is a dynamic parameter, whose value is changing at runtime, the optimal setting of \(upper\_bound\) is also dynamic and cannot be solved statically through models in Eq. (4)-(6). A rule of thumbs is to set the \(upper\_bound\) between 80% to 95%.

With \((\lambda-\lambda_{img})\) arrival rate of requests, not all requests will be scaled adaptively, and instead simply processed by default behaviour. This effectively reduces CPU utilization when high request loads cause CPU saturation.
At the same time, we need to tune the thread pool size of the QueuedExecutor to an optimal setting for this sampling the frequency to achieve better response time for $R_{img}$. After the CPU drops below peak load, the default settings need to be reset.

Figure 14 The open queueing network model of the image processing adaption

Therefore two action-based policies [17] are defined for controlling the CPU usage:

**Policy One:** IF (CPU usage $\geq$ upper bound) THEN (tune the frequency and thread pool size)

**Policy Two:** IF (CPU usage $<$ lower bound) THEN (reset the original settings)

As the application server also contributes to the CPU usage, it is not possible to simply solve the frequency value by Eq. (4) and (5). Therefore, to simplify the analysis and not introduce extra sensors and monitors for collecting and analyzing the application server performance details, we make another approximation as follows to set the condition for changing the frequency.

We can see that $R_{img} \geq (D_{img} + D_{adp})$, and $R_{img}$ can increase to several factors of $(D_{img} + D_{adp})$ when the CPU usage increases. So we can consider increasing the frequency settings (the default frequency is 1) under the constraints that $R_{img} < 2(D_{img} + D_{adp})$. We use the algorithm listed in Figure 15 to first find the frequency level and then determine the thread pool size under this frequency level that can meet the condition, $R_{img} < 2(D_{img} + D_{adp})$.
Figure 15 Algorithm for finding values of frequency and thread pool size

5.4 Developing ASF adaptive components

Improving the throughput and reducing the response time of the image server relies on two aspects of adaptation: (1) adapting the quality and resolution of the images requested; (2) configuring the frequency and concurrency level so that the CPU is not saturated.
This first adaptation trades off image resolution and quality against image processing time and image size to reduce the transport overhead over network connections. Based on the above analysis, we can use ASF components to create a control loop that determines how an image is to be scaled, as shown in Figure 16. The name of each component indicates the type of basic ASF component it inherits from.

- **ImageScaleEngine** is responsible for bootstrapping the adaptive engine and managing all the other elements.

- **BandwidthSensor** intercepts a client request and detects the client connection network speed \( B_{\text{network}} \) and request arrival rate \( \lambda \). 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 analysis model represented in formulas (1-3).

- **ImageScaleMetricsRepository** takes the output of the **ImageScaleAnalyzer** and stores a record of the resolution and quality for the image to be scaled. Each record is identified by a compound key with the invocation \( id \) and image file name.

- **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 analysed.
Figure 16 The control loop for scaling an image

These components interact using the asynchronous communication model described in section 4. Figure 5 earlier depicted an example of the communication between BandwidthSensor and the ImageScaleMonitor.

For the second adaptation, 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 into 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 17.

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*\(^2\). The *ConfigurationEffector* is associated with Policy 1 and Policy 2. It is responsible for setting the frequency and thread pool size of the scaling engine elements when the CPU is overloaded or resetting the default configuration when the CPU is below lower bound utilization. We have developed a Queuing Network model in section 5.3 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.

![Figure 17 The control loop for CPU usage](image)

5.5 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. Lifecycle

---

\(^2\) Only the default settings that have been changed are reset.
operations such as *startup* and *stop* can be invoked through the JMX console as shown in Figure 18.

![JMX console for The ImageScaleEngine](image.png)

Figure 18 JMX console for *The ImageScaleEngine*

The *BandwidthSensor* and *CPU Sensor* 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 Windows XP performance counters through the Win32 API.

The *BandwidthSensor* and *ImageScaleEffector* are deployed as JBoss interceptors [14], so that the signature of an invocation request and its return value can be captured. The screenshot in Figure 19 shows the configuration of *BandwidthSensor* and *ImageScaleEffector* for introspecting EJB components in the JBoss deployment descriptor. As shown in the simplified code screenshot in Figure 19, the business logic of the EJB component remains unchanged. The sensor and the effector intercept the method invocation and process the request in the body of method `invoke`.
public class BandwidthSensor extends AbstractInterceptor
    implements Sensor, Serializable{
    //....
    public Object invoke(Invocation mi) throws Exception {
        //code for collecting the network connection speed
    }
    //....
}

public class ImageScaleEffector extends AbstractInterceptor
    implements Effector, MessageProducer, RepositoryClient, AutonomicExecutorClient{
    //....
    public Object invoke(Invocation mi) throws Exception {
        //code for setting the invocation parameters
        // with calculated resolution
    }
    //....
}

public class ImageProcessBean implements SessionBean {
    public byte[] getImage(String imageName, float minResolution, float maxResolution)
        throws RemoteException {
        ImageMetrics im = imgPro.getImageInByteArray(imageName);
        // ImageProcessor imgPro = new ImageProcessor();
        if (maxResolution == 0)
            bi = im.ImageByteArray;
        else bi = imgPro.getScaledImageByGetInstance(im, maxResolution, defaultQ);
        return bi;
    }
}

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.

Figure 19 Example of ASF component deployment

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.
5.6 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.

5.7 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.

We then compared the performance of the JBoss server with and without adaptive behavior. 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.
We first compared the response time of the JBoss server with and without adaptive behaviour given a fixed number of requests of 10 clients. The aim is to observe the performance benefit gained according to the image size. The results are shown in Figure 20 to Figure 22. For small images of size less than 10KB, 90% of response times are reduced from 185 milliseconds to 100 milliseconds with the adaptive behaviour enabled. For larger images the improvement is more significant. For images of size between 100KB to 500KB, 90% of the response times is within approximately 3.23 seconds with adaptive behaviour enabled compared to 90% of the response times within 5.15 seconds with adaptive behaviour disabled. For images of size between 500KB and 2MB, adaptive behaviours improve performance with 90% of the response times is within 33.1 second while standalone JBoss server without adaptive behaviours has 90% of the response times within 38.95 seconds.

(a) adaptive behaviour disabled  (b) adaptive behaviour enabled

Figure 20 Response time distribution for images of size less than 10K
To evaluate the performance under varying requests of images, we 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.

**Figure 21** Response time distribution for images of size less than 500K

(a) adaptive behaviour disabled  
(b) adaptive behaviour enabled

**Figure 22** Response time distribution for images of size less than 2M

(a) adaptive behaviour disabled  
(b) adaptive behaviour enabled
The entire test lasts for twenty minutes. The same workload is applied to the server application executing with and without adaptation. With adaptation, we also empirically compared adaptive behaviour with various settings of the upper_bound of the CPU usage as shown in Figure 23 (a)-(c). The JBoss server configuration is exactly the same for all the tests.

Figure 23 and Figure 24 show the client response time distribution for these two categories of tests. With adaptation enabled, 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. When setting different levels of upper_bound value of the CPU usage, we also observed variations of the adaptive behaviour in term of response time and throughput. 90% of response times are within 27 (31) seconds and the over throughput is 223 (201) if the value of upper_bound equals 90% (99%). An intuition is that under the test of heavy workload, most of the time the CPU usage is over 90%, so setting it to 80% always triggers the adaptation earlier. As 99% means almost saturation, the policy condition is more restful in triggering the configuration adaptation.

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.
Figure 23 Response time distribution with adaptive behavior enabled

(a) upper_bound = 80%

(b) upper_bound = 90%

(c) upper_bound = 99%
6. CONCLUSION

In this paper we present an architecture framework known as ASF for developing adaptive behavior for applications running on application servers such as J2EE. 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.

Some requirements are implicit. For example in our case study, controlling CPU usage is not explicit in the requirements. These implicit requirements must be identified and represented by a policy description.

Developing adaptive applications 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 into grid computing environments with the support of 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

   http://www.oasisopen.org/committees/download.php/14347/wd-wsdm-primer-17.doc
Figure 25 Metrics of scaling images. The image size after scaling depends on both the resolution and quality. The higher the resolution or quality the larger the final image size scaled. The scaling time depends mostly on the resolution. The effect of quality on image scaling time is less significant.