Implementing Adaptive Performance Management in Server Applications

Yan Liu\(^1\) and Ian Gorton\(^2\)

\(^1\)National ICT Australia, Australia
\(^2\)Pacific Northwest National Laboratory, U.S.A

Abstract

Performance and scalability are critical quality attributes for server applications in Internet-facing business systems. These applications operate in dynamic environments with rapidly fluctuating user loads and resource levels, and unpredictable system faults. Adaptive (autonomic) systems research aims to augment such server applications with intelligent control logic that can detect and react to sudden environmental changes. However, developing this adaptive logic is complex in itself. In addition, executing the adaptive logic consumes processing resources, and hence may (paradoxically) adversely affect application performance. In this paper we describe an approach for developing high-performance adaptive server applications and the supporting technology. The Adaptive Server Framework (ASF) is built on standard middleware services, and can be used to augment legacy systems with adaptive behavior without needing to change the application business logic. Crucially, ASF provides built-in control loop components to optimize the overall application performance, which comprises both the business and adaptive logic. The control loop is based on performance models and allows systems designers to tune the performance levels simply by modifying high level declarative policies. We demonstrate the use of ASF in a case study.

1. Introduction

Application servers provide 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 a suitable platform for high performance and reliable software systems. However, such systems remain challenging to construct as they usually operate in dynamic environments with rapidly fluctuating user loads and resource levels, and unpredictable system faults.\(^4\).

Application server technologies enable administrators to tune application performance in response to functional and environmental changes that occur after application deployment. Such responses are made by adjusting configuration parameters, such as the server thread pool size, transaction time-out duration, and the maximum number of concurrent connections. However, deciding on the ‘best’ parameters settings is notoriously difficult, and must be performed through observation and experimentation every time an environmental change occurs. This results in on-going, high administrative overheads and costs for managing the system\(^1\)\(^4\)\(^9\).

A solution is to apply autonomic computing technologies to automate the performance management at runtime. Autonomic computing encompasses a grand vision of creating computer systems with properties of self-configuration, self-healing, self-optimization and self-protection\(^10\). In the realm of autonomic computing, application servers are an autonomic element that interacts, regulates and adapts its behavior in response to a wide range of changing environmental circumstances under administrative policies.

Performance and scalability are critical quality attributes within the scope of self-optimization and self-configuration. However, common performance engineering approaches usually do not aim to automate performance management at the run-time, but focus on building performance models from software artifacts\(^2\). Few efforts have addressed the issue of efficiently acquiring and reasoning with dynamic performance data during execution. In order to enable performance optimization of server applications at runtime, applications need to be extended with adaptive behavior implemented using autonomic computing technologies.

Following autonomic computing principles, adaptive behaviors for performance optimization of server applications utilize a control loop to constantly monitor the system and identify and handle events according to defined policies. This however creates new development and management challenges for application designers, namely:

- Construction of predictive models: given inputs that characterize the current execution environment, models must be solved to produce optimal parameter settings for controlling various aspects of application configuration and behavior.
- Separation of concerns: application business logic should be completely separated from the adaptive control logic for performance management. This promotes ease-of-change to the business logic, and supports a clear division of labor between application developers and adaptive control logic developers.
- Overhead: Environmental changes can occur rapidly, and hence responses, in terms of behavior adaptation, should occur with low latency. This indicates the
necessities of extra control loops to manage the resource consumption caused by adaptation so that it does not adversely affect overall application performance.

We have developed the Adaptive Server Framework (ASF) built on standard middleware services [5] for creating adaptive servers applications. ASF provides a component model for developing adaptive control components, and associated services that support the deployment of adaptive components with J2EE application servers. The details of the ASF architecture and implementation of its component model are described in [5], which includes a case study to evaluate the architecture in terms of the performance overhead incurred by ASF components. The empirical results demonstrate the performance improvement gained by using ASF and show that the design and implementation of ASF are lightweight.

In this paper, we focus on an approach to building adaptive performance management for server applications using ASF. It further discusses how ASF integrates techniques from performance engineering, software architecture and autonomic computing. The case study in [5] is explored in detail to illustrate the approach. Although our case studies are based on J2EE techniques, the framework can be implemented in other application server techniques such as .NET.

2. Related Work

Automating performance management has been an active area of research over the last few decades. The Software Development Life Cycle (SDLC) based approach towards performance management aims to utilize software performance analysis techniques within software engineering methods during the various stages of the SDLC such as analysis, design, development and testing. In early phases of the SDLC, performance annotations are created and refined as part of the architecture design artifacts to predict software system performance [1]. The role of the performance model is to define relationships amongst various control parameters of a software infrastructure element such as buffer pool size, connection pool size, and thread pool size and various performance measurements. Balsamo et al. provide a comprehensive survey of performance model based performance prediction techniques in [16].

Design-time performance prediction techniques guide the construction of an application for high-performance. However, if the application’s actual workload varies considerably from that expected during design, then the resulting performance cannot be predicted, and may in fact not be adequate. A solution is to incorporate mechanisms in to the application to automate performance management at run-time. This requires embedding adaptation capabilities within the application architecture to perform time-critical monitoring, analysis and enforcement of performance requirements.

There is considerable current research focusing on performance management using self-configuration techniques. These aim to ensure that a system operates within its required performance service levels by handling changes to the system’s run-time dynamics via reconfiguration (or tuning) of its key control parameters [12][17]. The basic building blocks of a self-configuring system are a performance model, a controller and a set of sensors and effectors [10].

A performance model relates the configuration information of various application components to workload profiles and settings for control parameters that can be used to adjust application performance. A controller component is responsible for implementing the control loop and adjusting the values of configuration parameters to bring the state of the system towards optimized performance. Sensor components monitor runtime behavior of the system and efficiently acquire performance related information. The control loop enacts its configuration or tuning tasks through effector components.

The implementation of sensors, effectors and the controller in an adaptive server application has a direct impact on the performance optimization technique to be used. First, as Menasce discussed [12], traditional performance modeling techniques are aim at strategic performance management goals such as capacity planning, while short-term tactical performance management requires an high-performance mechanism that can react within seconds (or less) to environmental changes. Second, the application business logic must be completely separated from the implementation of the control loops, so that the performance influences of the adaptive logic can be carefully controlled.

The Rainbow project [3] proposes an architecture-based approach with the emphasis on adaptive strategies and techniques for detecting architectural styles at runtime. The Rainbow project and our approach have architecture design principles in common, namely that adaptation should be modular and separated from the managed application, interacting with the application in a non-intrusive way. In Rainbow, adaptation is predefined based on the architectural styles of the system, while in this paper we address how application-specific adaptive behavior can be incorporated into existing systems driven by models. ASF embodies the systematic approach encompassing software engineering disciplines for developing adaptive behavior for performance optimization of server applications.

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 adaptive logic should be encapsulated into components external to the application implementation.

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 executing a different algorithm or setting the server’s configuration parameters to fulfill business goals.

ASF provides a framework for developing adaptive server applications that continually monitor and optimize their performance. The framework embodies principles and techniques from performance engineering, software architecture and autonomic computing technologies. The following sections explain how ASF application development leverages these areas to build adaptive applications.

### 4. The Approach

We consider the adaptivity in terms of optimizing the performance at runtime. The behaviour to fulfill performance goals is implemented by adaptive control logic. The control logic implements a performance model that, given inputs to characterize the current execution environment, solves the model to produce optimal parameter settings for controlling various aspects of application configuration and behavior.

#### 4.1 Construction of performance models

There are three aspects to consider when devising a performance model for adaptive behavior of server applications:

1. Convert descriptions of performance goals into quantitative and measurable metrics. These measures become the target outputs of the performance analysis.
2. Determine the impact of adaptive behaviors to the server applications. Issues include how to model the performance related behavior of adaptive components and their interactions, especially for interactions with the underlying server infrastructure.
3. Calibrate application specific factors such as parameters into the performance models and populate their values. For example, tuning the thread pool size of J2EE application servers is a generic self-configuration strategy for all applications running on an application server, however, optimal cache size tuning is only needed for applications that actually cache data at the application server layer.

The above approach gives a strategy for devising models for online performance analysis using performance engineering techniques. The approach categorizes dynamic environmental variables to three groups, namely, application specific control parameters, server configuration parameters, and resource constraints parameters:

- **Application specific control parameters** quantify the adaptive behavior. For example, an application could adaptively compress large XML documents that must traverse slow network connections to save network bandwidth. The factors related to compression, such as the final compressed file size, the compression compute time and the network connection speed are specific to this application.
- **Server configurations parameters** refer to environmental variables such as the thread pool size, the number of database connections, buffer sizes and so on.
- **Resources constraints parameters** qualify the high level administrative policies that the adaptation should meet at runtime, such as the CPU usage should be kept within 95% to prevent the system being saturated.

Our approach employs two performance models to handle these parameters, namely, **system models** and **control models**. The dependency between the models and parameters is shown in Figure 2, where a solid arrow represents dependency, while the dashed arrow indicates controls.

![Figure 2 Performance and adaptiveness relationship](image-url)
This model also depends on the resource constraints specified in the high level administrative policies. Traditional offline performance modeling theories and techniques, for example, queueing network models, can be used to build the system model.

The impact of adaptive behaviors on performance also has to be captured, because this impact may affect server configuration parameters and the resource utilization. For example, dynamic caching may reduce the computing overhead of access to a database and therefore the cache size parameter can affect the service response time and CPU utilization.

The control model is used to capture the relationship between adaptive behavior, server configuration and resource utilization. The control model accepts application specific control parameters as inputs and its outputs guide the adjustment of server configuration and resource utilization. Control models can be built from various formalisms dependent on the application context, such as utility functions, models of control theories and solutions from complex algorithms.

Note that the system models and control models are not isolated from each other, but are connected by the server configuration parameters, resource constraints that the adaptation might affect and application specific control parameters. Ideally, the control models and system models can be automatically built online by monitoring the environment and learning the system’s behavior. This is an active and open research topic and out of the scope of this paper. Some research work can be found in [17]. In our framework we require the performance models to be built offline based on observations. A similar approach has been applied in [8] [12].

4.2 Construction of control loops

Once the performance analysis aspects are depicted in the model, the next step is to design and implement the components as ASF control loops.

A generic reference framework for control loops has been proposed by IBM’s Autonomic Computing initiatives [6]. As illustrated in Figure 3, a control loop consists of tasks for monitoring the execution of the application, analyzing the data collected, planning the necessary responses based on policies that govern adaptation, and executing actions to enforce the adaptive behavior.

\[\text{Figure 3 Control loops in generic autonomic computing architecture}\]

ASF implements this generic architecture for online performance management, as is shown in Figure 4. In ASF a number of component types are provided, namely:

- A control loop is driven by the performance related knowledge of the system under management. Performance knowledge can be as simple as a real-time value of a parameter, or regression results of a series of monitored data. Performance knowledge is quantified as control or configuration parameters or resource constraints whose values can be monitored or reasoned.
- Sensors are components that provide probes to intercept invocations, detect environmental measures, and collect data by sampling without analyzing them, such as the CPU usage, network connection speed, arrival rate of requests, and memory usage.
- A monitor component composes a detailed view of the system's states or metrics for further analysis. Samples data from sensors are further aggregated, correlated or filtered by monitoring function.
- An analyzer component utilizes system and control models to observe and analyse situations to determine if some change needs to be made. The analysis is influenced by performance knowledge updated at runtime.
- A plan component determines the behavior of control components according to the policies specified. A policy is a representation of desired behavior or constraints on behaviors defined in a standard external form [14]. For example the resource constraints can be specified in policies.
- The execute component provides mechanisms to schedule the execution of control components, such as concurrency management and bootstrapping all the control components.
4.3 Standard-based architecture design

There are three design issues of the architecture that implements the proposed approach in section 4.1 and 4.2: (i) identify the mechanisms to implement sensors and effectors; (ii) design communication pattern to compose these components; (iii) achieve non-instrument performance adaptation so that no changes to the application business logic or server infrastructures are required.

The proposed solution is to introduce a standard-based management layer between adaptive control components and the hosting application server as shown in Figure 4. Our implementation is based on J2EE technologies, but the solution is applicable to other application server technologies.

The management layer includes utilities and mechanisms of two categories. One is to monitor the runtime behavior of the application and the underlying platform; the other is to reconfigure the settings at different levels. J2EE JMX (Java Management Extensions) based management architecture is leveraged in this architecture [15] for both categories. 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 [15].

JMX also provides mechanisms for sensors to collect data on the application server’s environment, and for effectors to tune the server configuration or to change the server’s behavior through the management layer interfaces. There are two ways of implementation, dependent on the architecture of the application server. One is to deploy sensors and effectors as customized Interceptors or Request handlers, which are implemented by many application servers to process requests and responses. The other way is to wrap sensors and effectors with proxies and reflection techniques to instrument the requests and responses.

Individual components are connected to form a control loop. The execution of control components is driven by the messages generated at runtime. Therefore message-based asynchronous communication patterns helps form loosely coupled control loops to performance management. Connections between components can be dynamically formed. For example, using publish/subscribe pattern, messages from one component can be published to a specific topic, and subscribers of that topics can receive that message and process it.

The advantage of this architecture is that it minimizes the dependencies of performance adaptation control components on specific features of the underlying application server. The application server, the management layer, and the adaptive components together form the architecture for augmenting applications with adaptation. Applications are not aware of the existence of adaptive components. Thus, development concentrates on the business logic implementation. Moreover, the separation between application, server and adaptation implementation is especially useful for introducing performance adaptation into existing applications, of which access to the source code is not practical. Thus, the cost of transformation and maintenance is largely reduced.

5. Case Study

In this section we explain how to apply our approach to constructing adaptive behavior of an image retrieval application using ASF. The case study is motivated by workload characterization studies of web sites which indicate 80% of all requests were for image files. As access from mobile devices with limited display capabilities and low bandwidth connections becomes prevalent, image transfer times can cause performance problems.

The case study aims to improve the response time of an image server. This is achieved by scaling the image size and resolution adaptively according to the network connection speed, the resolution requirements from the requests (for example, some mobile devices can only display images with limited resolution) and the server resource utilization.

An overview of the image retrieval application is illustrated in Figure 5. A client sends a request to the application server for a specified image containing a minimum and maximum resolution for the image. By default, the maximum resolution is unit 1, 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 processing application just scales the images to the maximum resolution requested.
5.1 Construction of control model

The performance goals of this image server application are to improve the throughput and reduce the response time of the application 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.

In order to find a relationship between metrics such as the scaled image size, scaling time, and image quality, we examined empirical measurements from scaling 100s of images with sizes from 1KB to 2MB. The empirical results show 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 the image scaling time affects the application’s response time, and the scaled image size determines the delay of transferring the images given the network speed. Based on this preliminary analysis, the dependency between these characteristics is shown in Figure 6. In order to simply the analysis, here we do not consider the dependency between resolution and quality. Based on this analysis, 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.

This description helps to establish the control logic to represent this adaptive behavior. According to the approach we described in Figure 2, we can capture performance related parameters of this specific application as shown in Figure 6. The control logic decides the values for the application specific control parameters, namely the image size, the resolution and quality of images. Figure 6 clearly shows that the performance metrics of interests depend on these control parameters.

Designing the control components to fulfill this control logic and capture the dependency of parameters requires a control 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.

We make an assumption that resolution is a function of the CPU usage of the application server, so 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 is 100%, the resolution is 50% of the original size, giving \( a \) as approximately 0.6931. Figure 4 shows this simple model for representing the relationship between CPU usage and resolution.
In order to satisfy a client’s resolution requirement, the actual resolution \( Y \) used is determined as

\[
Y = Y_{\text{min}} \text{ if } Y_r < Y_{\text{min}} \text{ or } Y = Y_{\text{max}} \text{ if } Y_r > Y_{\text{max}} \tag{2}
\]

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 (2). The higher the network connection speed, the higher the quality image can be delivered.

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

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 speeds 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 8.

5.2 Construction of the system model

The CPU cycles used by the adaptive engine that starts and manages all the adaptation components, and the application server are major sources of CPU resource consumption. They can be modeled in a open queueing network model with two load independent multiple servers, one for the image processing engine and one for the application server.

![Figure 9 System model using queueing network model](image)

In Figure 9 \( m \) and \( n \) represent the thread pool size of each respectively, \( \lambda \) and \( \lambda_{\text{img}} \) represent the request arrival rate at the application server and the image processing engine respectively. \( \lambda_{\text{img}} \) depends on the sampling frequency \( f \), for example \( f=1 \) means doing adaption for every client request, and \( f=3 \) means sampling every third client request.

\[
\lambda_{\text{img}} = \frac{\lambda}{f} \tag{4}
\]

Applying the MVA algorithm for an open queueing network model [10], we can derive the CPU utilization and response time of the adaptive engine below

\[
U_{\text{img}} = \frac{\lambda_{\text{img}} (D_{\text{img}} + D_{\text{adp}})}{m} \tag{5}
\]

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

From Eq.(4)(5), it is clear that to reduce the CPU usage when it exceeds the upper bound threshold, such as \( U_{\text{img}}>95\% \), we can increase the frequency value to reduce the arrival rate \( \lambda_{\text{img}} \) in the adaptation engine. At the same time, we need to tune the thread pool size of the adaptation engine to an optimal setting for this sampling frequency to achieve better response time for \( R_{\text{img}} \). After the CPU drops below peak load, the default settings need to be reset.

5.3 Construction of control loops

The performance models discussed in section 4.1 and 4.2 trade off the image resolution and the quality against the image processing time and the image size to reduce the overhead of the transport over network connections. Based on the above analysis, we can now develop components to create the control loop that determines how an image is to be scaled, as shown in Figure 9.

- \textit{ImageScaleEngine} is responsible for bootstrapping the adaptive engine and managing other
components, such as their execution concurrency. It acts as the execute component.

- **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 performance model represented in formulas (1-6) and the (MVA) mean value analysis algorithm in [10] for finding optimal values of frequency and thread pool size.

- **ImageScaleMetricsRepository** takes the output of the ImageScaleAnalyzer and stores a record of the resolution and quality for the image to be scaled and the optimal values of frequency and thread pool size. 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 analyzed.

Figure 10 The control loop for scaling a message

5.4 Architecture implementation of control loops

We implemented each adaptation component involved in control loops of ASF using pure Java objects. Each component has been implemented with well defined interfaces. In this case, sensors and effectors implement JBoss interceptor interfaces and interact with JBoss server through JMX in a non-instrument way.

Components interact using message-based communication pattern is illustrated in Figure 12. Message producers have the dispatchMessage() method that dispatches messages to the QueuedExecutor. The executor allocates a thread to iterate over all associated message consumer components and invoke their handleMessage() callback method to process the message sent to it.

ASF also provides default implementation of services for policy management, concurrency control and tools for deployment. Due to the space limitation, we omit the details of implementation. We believe the generic computing paradigm described in ASF allow this case study to be implemented by other autonomic computing technologies, such as IBM’s autonomic computing toolkit [7], this also remains our future work.

6. Lessons Learnt

Our experience in building adaptive behavior for the image server application has provided some insights into developing adaptive server applications. The design and implementation of adaptation can be difficult for even moderately complex applications. We summarize the lessons learnt as follows:

- Understanding application specific adaptation requirement

The adaptive logic design and implementation are driven by the application’s requirement for adaptation. The high-level requirements need to be translated into control mechanisms to fulfill quality attributes requirements, for example performance in this paper. This requires a detailed understanding of both the application and environment characteristics for the quality attribute of interest. The application and the environment can interact in a very subtle way, and the relationship or dependencies among the quality attribute related factors are not obvious. Therefore empirical observations and abstraction of the observations are
vehicles necessary to distill key factors and their relationship.

- Developing analytical models
  Performance models play a critical role in reasoning over the system’s behavior and providing the necessary rules for adaptation. Performance models are embedded in the implementation of control components and are part of the control logic. Developing accurate performance models demands performance model expertise, which is not commonly available from the developers of adaptive behavior. This creates an extra barrier for implementing adaptive server applications. It is an open research questions how automated performance model techniques [16] can facilitate the development of performance models for adaptive applications.

- Programming and support
  Developing control loops is difficult. Designers need a programming model to solve many of the low level problems of interacting with the application and performing efficient analysis and adaptation. Fast and non-intrusive implementation is critical to build efficient adaptation. We believe that developing control loops is a specialized task, and hence solutions should shield application programmers from the inherent complexity. There are several toolkits available for developing adaptive behaviors with commonly used components and web services. However there is no quantitative or qualitative evaluation of these toolkits, so it is totally up to knowledge of the developers to decide which programming model is appropriate.

- Testing and maintenance
  Testing and debugging adaptive behavior implementation is very challenging. This is because certain behaviors are only triggered under specific conditions, which quite often have to be simulated. When exceptions occur, it is also difficult to trace the execution of components or services especially when they are distributed. This indicates interdisciplinary research is needed between the adaptive application development and software testing and debugging communities.

- Evolution
  The current implementation of adaptive behavior is deployed onto the infrastructure using very platform specific techniques. For example, the adaptive image application is deployed on JBoss using its specific invocation techniques. Once the infrastructure evolves and new changes occur in the infrastructure, adaptive components with platform specific deployment might not be able to work and the deployment migration between platforms has to be done manually. A potential solution is that the deployment is specified in a descriptor and tools supporting the migration between different infrastructures can be used.

7. Conclusion

In this paper we present an approach and associated technology for developing performance driven adaptation of server applications. The approach provides a strategy for devising models and control components for online performance analysis. By using ASF, the implementation and deployment of control models are integrated with software architecture design and autonomic computing technologies. We demonstrate the use of our approach in building an adaptive image server application. The lessons we learn from this work are summarized on different stages of developing adaptation, design, implementation, testing and maintenance. They present open research questions and also setup the scope for our future research.

8. Acknowledgements

National ICT Australia is funded through the Australian Government’s Backing Australia’s Ability initiative, in part through the Australian Research Council.

9. References

Autonomic Software (DEAS’05) Workshop, St. Louis, USA, May 2005.


