Composing Adaptive Web Services on COTS Middleware

Yan Liu\(^1\) and Simon Truong\(^2\) and Shiping Chen\(^3\) and Liming Zhu\(^1\)
\(^1\)National ICT Australia, Alexandria, NSW, Australia, 2015
\(^2\)University of New South Wales, Australia
\(^3\)CSIRO, Sydney, Australia
jenny.liu@nicta.com.au, smht758@cse.unsw.edu.au, shiping.chen@csiro.au, liming.zhu@nicta.com.au

Abstract

Composing adaptive and self-managing Web services needs plug-and-play architecture so that the deployment of control components does not require changes made to the Web services and the host middleware platforms. This is especially challenging for Web services running on COTS middleware platforms, such as Microsoft .Net. In this paper, we propose an architectural solution that introduces a management proxy between adaptive control components and Web services. The management proxy can be customized and seamlessly integrated with a COTS middleware platform by leveraging the existing middleware mechanisms. This solution enables dynamically composing adaptive Web services on COTS middleware without stopping its services. We demonstrate this architecture by a realistic Web service application built on .Net Windows Communication Foundation (WCF). The performance overhead incurred by this architecture is measured, and the results validate that our solution is efficient in terms of performance and flexibility.

1 Introduction

Dynamic adaptation and self-management of Web services rely on a set of adaptive components, which are composed into control loops for constantly monitoring the system, analyzing the situation and deciding on the actions to affect the system’s behavior [13, 16]. These adaptive components are generally concerned with quality attributes such as performance or dependability.

IBM’s Autonomic Computing initiatives [8] proposed a generic architecture to compose such adaptive components. One basic principle of this architecture is that adaptive components should be separated from the business logic implementation of Web services. Such a separation of concerns raises several issues at the software architecture level, including:

1. Plug-and-play: adaptive components should be deployed to the Web service environment without requiring modifications made to the Web services.

2. Hot-swapping: adaptive components should be updated and replaced without shutting down the operation of Web services.

3. Performance: the computing overhead of adaptive components should be optimized to avoid adverse performance degradation of the overall application.

4. Distributed deployment: adaptive components can be deployed in a highly distributed environment.

Realizing such an architecture imposes challenges. One of the remaining issues is that the architecture implementation relies on the middleware mechanisms (inside Web or application servers) to implement and deploy these components. However, COTS middleware platforms preclude the assumption that the middleware platform can be modified to provide desirable mechanisms. Even shutting down its current services is an unrealistic assumption for deploying new adaptive components. Hence it is essential that an architecture solution should ensure dynamic composition of adaptive components and the integration to the middleware.

In this paper, we design an architecture framework to compose adaptive components with Web services. In this framework, a management layer is introduced between Web services and the adaptive components. The management layer works as a proxy, coordinating the information flows from Web services, to the middleware platform and to adaptive components. The innovation of this approach is that it enables the COTS middleware as a self-contained environment hosting business Web services and adaptive components coherently, without changing the middleware libraries or the Web service implementation. This is achieved by mapping the architectural concepts to underlying middleware mechanisms, including message interception, behavior extension, and service configuration.

The structure of the paper is as follows. Section 2 discusses the related work. Section 3 proposes the architecture
framework. Section 4 illustrates the utilization of this architecture in different scenarios, including a realistic Web service application running on .Net Windows Communication Foundation (WCF). Section 5 evaluates the efficiency of this architecture by observing and measuring the performance characteristics of this architecture. This paper concludes with reflections and future work in section 6.

2 Related Work

The related work are discussed from three aspects, adaptive and self-managing architectures, feedback control development, and mechanisms for composing adaptive systems. Many of the related work actually cover all of these three aspects but with a different focus of each.

Applying feedback control principles to support different service classes on the Web has been investigated in recent literature. Lu et al. [12] developed feedback control loops for service delay guarantees and embedded them into the Apache Web server. Diao et al. [5] used the multi-input-multi-output control to achieve desired CPU and memory utilization on a Web server. These approaches share the common limitation that the source code of the Apache Web server was modified to add new libraries of the control loops, which limited their application to COTS middleware platforms when the access to source code is not available. In our paper, we focus on the issues of integrating adaptive and self-managing controls with middleware platforms without modifications made to the source code. We introduce a management layer working as a proxy between Web services and the adaptive control components. A similar approach was realized by Liu et al. [10], in which an open source proxy application was modified to add adaptive components. The resulting proxy application was then connected with the Apache Web server. The difference is that our solution does not require external software but relies on the existing mechanisms of the middleware itself.

A different approach to adaptively changing the structure of the applications was developed by Dashofy et al. [3]. This work applied xADL, an architecture description language to express and execute a repair plan for healing a system’s architecture at runtime. This approach was supported by a modeling tool suite, called ArchStudio. The architecture descriptions of components and connectors in xADL were mapped to the running systems. The repair plan was realized by changes made to the software architecture, which was represented as architectural differences. The differences served as an architectural patch. The results of applying such a patch could be analyzed before deploying it to a running system. This approach aimed at predicting the impact of architectural changes by “what-if” analysis before running the self-managing operations. This approach was useful in preserving safe changes made by adaptive and self-managing controls.

The paper by Martin et al. [13] described the development of autonomic Web services following the Web Services Distributed Management (WSDM) standard. The work focused on using WSDM as it pertains to interfaces between existing sites under autonomic management. Each site was associated with autonomic managers developed by the authors’ previous work. This paper addressed the composition problem at a higher level than our work, while our work is focused on composing autonomic management for individual Web services.

A comprehensive survey [14] discussed existing technologies that could enable dynamic composition of adaptive software. It also classified different approaches according to how, when and where composition may occur. The core of all these approaches was intercepting and redirecting interactions among program entities. These mechanisms help to customize our architecture framework to a specific middleware platform. We demonstrate later in section 3.2 the utilization of .Net WCF mechanisms to address the needs of intercepting and redirecting messages.

3 The Architecture

In this paper, we consider the architecture scope in terms of the adaptation types the architecture should support, namely architecture adaptation and behavior adaptation. Architecture adaptation monitors its software architecture model to create changes in the structure of components and their interactions; while behavior adaptation changes the execution of components by changing the configuration of components or by monitoring and intercepting interactions between components. An efficient architecture to address both types of adaptation should ensure the following requirements:

Req1. Interception of incoming requests: the ability to intercept messages and extract the message context.

Req2. Modification of outgoing responses: the ability to intercept and modify the responding messages.

Req3. Dynamic deployment of adaptive behavior: the ability to enable the transparent deployment of adaptive components at runtime. The business logic or the middleware platform should not be modified as a result of the deployment.

Req4. Asynchronous communication: the ability to support both synchronous and asynchronous communication. For example, a sensor can keep feeding itself with data without being blocked by the computing of an analyzer after the sensor sends off the data to the analyzer.

Req5. Singleton services: in order to maintain shared state of adaptive components, singleton services should be supported in a distributed environment.
Req6. Extensibility of adaptive behavior: throughout the application’s life-time, the adaptive and self-managing concerns may change as the system evolves. This requires the architecture to be extensible and flexible enough to dynamically add and update adaptive components.

3.1 Conceptual Architecture Framework

The architecture that addresses above requirements comprises of three layers, namely the layer of business Web services, the management layer which hooks directly into the underlying middleware platform, and the adaptive component layer which encapsulates the adaptive behavior. The generic architecture framework is depicted in Fig. 1. We discuss the key elements of each layer in details.

The adaptive component layer encapsulates the control logic. In this paper, we summarize the common functionality of adaptive components [9, 14, 16] as follows:

- A sensor is a component that provides probes to detect environmental measures and feed the collected data to other adaptive components such as an analyzer.
- An analyzer is a component comprising models and rules for use in implementing the required adaptive behavior.
- An effector makes adaptive behavior occur.

According to the above requirements, these adaptive components should be exposed and accessed as Web services if distributed. Thus, their deployment should be separated from the Web service business logic in order to achieve dynamically adding and updating adaptive components. In addition, the singleton design for adaptive component is necessary to share the state of individual adaptive components. For example, sensors counting the arrival requests should maintain a singleton instance to guarantee the consistency of the counting.

The management proxy is responsible for tapping into the operation of the service host and acts as the proxy between the Web services and the adaptive components. This is done by a dispatcher that routes the intercepted message to its destination. The dispatcher can be programmed with fairly complex behavior rather than simple condition based actions. For example, the dispatcher can be integrated with a policy framework, which specifies an action dealing a message. This will be covered in more details in the next section of architecture mapping to a specific middleware platform.

When a request arrives at a service, the management proxy intercepts the request. That request is routed by the dispatcher to a sensor component for an inspection and data collection. This function of the management proxy is defined in the component Incoming Request Handler. The data collected by the sensor is then passed to an analyzer component that determines the proper adaptive actions to take. Any consequent actions will be carried out through an effector component that is invoked by the component of Outgoing Response Handler in the management proxy.

3.2 Architecture Mapping

The conceptual architecture needs to be realized on a particular middleware platform. This task includes the mapping of the elements and associations in Fig 1 to the middleware components and communication mechanisms respectively. In our previous work, we have applied this architecture to develop several different adaptive applications on various popular open-source middleware platforms [7, 11, 17]. The common characteristics shared by these middlewares is that the architecture supports a pipeline of interceptors or handlers to methods of the target class instance. This allows adaptive behaviors to be triggered from the inside of interceptors or handlers without changing the code. This solution enables the plug-and-play architecture to integrate adaptive components with middleware and separate their code with business logic components.

We consider a COTS middleware Microsoft Windows Communication Foundation (WCF) platform as the candidate middleware to illustrate this architecture. WCF provides the essential components and mechanisms to enable this architecture, including services, contracts, behaviors and process host. Table 1 summarizes the relations of WCF mechanisms, architecture components and the requirements listed above. Detailed technical descriptions of these components can be found in [15].

<table>
<thead>
<tr>
<th>WCF Mechanism</th>
<th>Architecture Component</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Behavior</td>
<td>Management proxy</td>
<td>Req.1,2,3</td>
</tr>
<tr>
<td>PIAB</td>
<td>Dispatcher</td>
<td></td>
</tr>
<tr>
<td>Host process</td>
<td>Service host</td>
<td>Req.3,6</td>
</tr>
<tr>
<td>Service</td>
<td>Management proxy</td>
<td>Req.3,6</td>
</tr>
<tr>
<td></td>
<td>Adaptive components</td>
<td></td>
</tr>
<tr>
<td>Contract</td>
<td>Management proxy</td>
<td>Req.4,5</td>
</tr>
<tr>
<td></td>
<td>Adaptive components</td>
<td></td>
</tr>
</tbody>
</table>

We identify another .Net framework Policy Injection Application Block (PIAB) [15] and combine it with WCF behaviors and services to implement the dispatcher and functionalities of the management proxy. PIAB supports the interception of messages by injecting handlers along the
path of message processing. Each handler can fulfill a certain task, such as logging, security checking, transaction and etc. Handlers are in a pipeline, forwarding messages to the next handler and returning messages to the previous one. Handlers are transparent to clients. PIAB uses InterceptingRealProxy components to delegate a message to the handler chain. If there is no handler, the Web service instance will be returned to the client. The resulting architecture mapping to WCF is shown in Fig 2. The sample code in Fig 3 illustrates how a handler is injected and instantiated. The message is intercepted by the handler AdaptiveHandlerInstanceProvider and the adaptive component is invoked in the method AfterReceiveRequest.

4 Example Application

4.1 Stock-Online Web Service

Stock-Online is a simulation of an online stock-broking system. It was developed in CSIRO as a benchmark application to evaluate middleware performance and scalability [6]. The application itself has been used as a means of evaluation in other research projects [1, 2]. Stock-Online is an illustrating example application to demonstrate our architecture. We present several scenarios to enhance Stock-Online by composing adaptive Web services.

Stock-Online supports six business transactions and enables users to buy or sell stocks, inquire about the stock prices, and get a holding statement detailing the stocks they currently own. The supporting database has features to track customers, their transactions, and payments. There are four database tables to store details for accounts, stock items, holdings, and the transaction history. Further details of performance evaluation using Stock-Online are described by Gorton et al [6].

The original Stock-Online was implemented as an J2EE application. In this paper, Stock-Online has been rewritten using .Net and C#. In this new implementation, each of the six key transactions has been wrapped as a WCF service. These services are hosted in IIS. We develop a .Net ASP Web client to consume these services.

4.2 Adaptive Performance Management

In the Stock-Online scenario, which may be subjected to an often volatile market place, an example of such a period might occur during a major market crash causing stakeholders to rushing into either buying or selling. Such periods of high strain, could cause the system performance to
diminish as a result of the increased strain on the computing resources. As an example, the proposed adaptive development hence aims to transparently monitor the system environment and improve performance using the classic admission control method. The purpose of this example is to illustrate the practical usage of this architecture, rather than improving the admission control methods.

The architecture design in Fig 2 is applied in this development. Fig. 4 shows the adaptive behavior in a sequence diagram. The implementation leverages the mechanisms listed in Table 1. Realizing the adaptive performance management requires the following adaptive components, including:

- CPU sensor, collecting CPU utilization;
- Workload sensor, counting the requests arrived and calculating their arrival rate;
- Analyzer, implementing the concurrency control rules and determining the optimal value of MaxConcurrentCall;
- Effector, modifying the value of MaxConcurrentCall;
- Manager, triggering adaptive behaviors given a set interval by invoking the analyzer to produce the analysis results and effector to update the configuration.

When a client request arrives, its incoming message is intercepted by the management proxy. The management proxy is composed of a handler AdaptiveHadler implementing two WCF interfaces IEndpointBehavior and IDispatchMessageInspector as described in section 3.2. The management proxy forwards messages to the dispatcher, which initializes the manager and invokes to the workload sensor. The manager triggers the adaptive behavior at a constant interval. Adaptive components interact concurrently to produce the optimal configuration of the server thread pool size, and finally the effector takes an action to either tune the pool size or enqueue requests when the pool is empty. WCF provides a default solution to enqueue requests. When the assigned limit of MaxConcurrentCall is met, consequent requests are automatically queued by WCF. This mechanism saves our engineering effort to realize a waiting queue associated with the adaptive behavior.

The workload sensor and the effector are the components that directly interact with the management proxy. All of these adaptive components are exposed as WCF services and deployed in the same IIS server that hosts the StockOnline Web services. As the workload sensor maintains the state of counters for arrival requests, it is implemented as a singleton service using the WCF contract option.

### 4.3 Adding New Control Components

The architecture in the previous section also supports dynamically adding and updating components. Fig. 5 illustrates how this works in a simple example. Originally, a client sends requests to the broker service of StockOnline. The broker further invokes an account service. Based on above discussions, the architecture is now supporting adaptive admission control. Assume that a new component Monitor is introduced to examine the message with a byte format attachment. A handler is added to the management proxy, examining the the message body and instructing the dispatcher to route messages either to the original sensor or the new monitor. This can be done by changing the configuration file of the management proxy service.

As a result, updating adaptive components is independent of the StockOnline business logic, and thus makes the architecture dynamically extensible. One limitation is that the service host needs to be restarted to take effect of the
new deployment. In the next section we evaluate the overhead incurred by restarting the adaptive services, such as the management proxy.

### Figure 4. Adaptive control sequences

![Figure 4. Adaptive control sequences](image)

4.4 Exception Handling

It is critical for adaptive components to ensure that changes are safe and do not incur any unfavorable consequences. The computing of adaptive components can throw exceptions and abort execution. The post-processing task needs to deal with the exception when it is caught. For adaptive components, typical exception handling mechanisms at the code level can do the post-processing. For example, the dispatcher can catch an exception when the instance of the adaptive component is null or the adaptive component can return a runtime exception to the dispatcher. An exception of the dispatcher can be returned to the last handler in the pipeline by adding it to the message that passes along pipeline. This allows handlers earlier in the pipeline to carry out their post-processing tasks as the invocation unwinds, such as creating and writing a log message. We recommend that an unhandled exception raised in a handler should be carefully avoided, otherwise the previous handlers will not be able to execute their post-processing tasks.

### Figure 5. Adding new control components

![Figure 5. Adding new control components](image)

5 Evaluation

We evaluate the performance of our architecture implementation from two aspects: the overhead incurred by adaptive behavior; and the performance of deploying the adaptive behavior. We setup a test bed to evaluate this architecture for both aspects.

The system under test (SUT) includes the Stock-Onilne Web services, adaptive components and the database server. They are deployed on two identical Windows XP machines with Xero Dual Processors 2.4GHz CPU, 3GB RAM. One hosts the Web services and adaptive components on .Net 3.0, and the other is running Microsoft SQL server 2003. The workload is generated using Microsoft Web Application Stress (WAS) tool. By changing the parameters of WAS (such as the interval between requests, the number of clients, and the percentage of requests to individual transactions), we can produce different patterns of workload for the Stock-Onilne services. The statistics of response time and throughput are collected by WAS. Each test is running for at least 5 minutes and repeated. The standard deviation for each performance metric is below 5%.

5.1 Delay by Adaptive Behavior

The overhead incurred by adaptive components includes the computing time of the adaptive components as well as the time spent on updating the configuration and reloading the service context. It is technically challenging to measure the exact computing time. This is due to the fact that communication between the management proxy and the adaptive components are driven by messages asynchronously. For example, as shown in Fig. 4 when the workload sensor is invoked, it returns immediately. Computing tasks of other adaptive components are triggered by the manager at a set interval. Our solution is that the time spent on management
proxy and adaptive components is measured separately and added up as an approximate estimation of the overall delay. We varied the arrival rate of requests from 10 to 100 requests per second, and observed that the delay remained constant, independent of the arrival rate. Hence, given the computing environment of the SUT, the approximate delay is $D = 1 sec$. In order to gain the benefit of adaptive behavior and trade-off the extra delay, the interval should be set larger than $D$, which is used by the manager component to trigger the adaptive behavior.

5.2 The Evaluation Results

Following the admission control method, we devised the adaptive components together with the implementation of the management proxy to enable the adaptive performance management on Stock-Online services. The empirical results in Fig. 6 show the performance with adaptive behavior compared to the original Stock-Online services without adaptive behavior.

The above graph in Fig. 6 shows the average response time measured for Stock-Online services in term of TTLB (time for last byte). It shows a slight improvement less than 5% when the workload is below 50 requests/second. This is because the system still has the capacity to handle the workload and the benefit of adaptive behavior is insignificant. As the stress level increases, the adaptive behavior helps to prevent the CPU being saturated by controlling the concurrency levels and queueing extra arrival requests. As a result, the performance margin gained by adaptive behavior increases to approximately 10% at 100 requests/second. The performance improvement demonstrates that the adaptive architecture is scalable as the workload increases, and the benefit resulting from a proper adaptive strategy (such as the admission control in our case) can trade-off its overhead incurred. The empirical results in the current evaluation validate our architecture solution in composing adaptive services.

The development of adaptive strategy or method is out of the scope of this paper. Other strategies for different quality attributes, such as reliability can also be applied in this architecture. This remains our future work to develop other examples to further validate our architecture.

6 Reflection and Conclusion

The architecture challenges of developing self-managed systems have been outlined by Kramer and Magee [9]. Of the many aspects of adaptive and self-managing architectural design, our research is focused on issues to achieve loose coupling and separation of concerns at the architecture level.

In this paper, we present our solution towards seamlessly composing adaptive components for COTS middleware-based Web services. By introducing a management proxy between the Web services and adaptive components, this solution effectively decouples the business logic in the Web services and the control logic embedded in those adaptive components. In addition, the management proxy leverages existing middleware platform mechanisms, and it can be customized for COTS middleware without changing the middleware platform implementation. As most middleware provides similar mechanics (such as interceptors and handlers) for extending their core functionalities to help custom and advanced development, the architecture is easy to be adopted for different platforms. Although we only demonstrate the usage on .Net WCF due to the space limitation, we have already validated this architecture on other Java Web service applications [4, 7, 11, 17].

Based on this architecture framework, our on-going research continues the investigation on issues of inter-component coupling for adaptive components. In this paper, those adaptive components have explicit logic dependencies. For example, the effector depends on the analyzer; the analyzer depends on the sensors; and the manager invokes the analyzer and effector. Their interactions are hard coded in the implementation. This tightly coupled implementation is simpler to implement, but this comes at the cost of flexibility and scalability especially for large scale, complex systems with multiple control logics.

A better design is for adaptive components to be identified by their logical or functional roles, so that connections between components are not hard coded, but established dynamically by their intent or roles. The resulting architecture will be much scalable and useful in addressing the challenging issue of preserving safe changes for self-managing architectures. For example, misbehaving adaptive components can be more easily isolated and replaced than those with hard code references. This requires the system to include a repository and the discovery mechanism, or similar functionality that the adaptive components can use to locate and establish a connection with the appropriate component(s) that fulfill the required service.

7 Acknowledgment

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

References


Figure 6. Performance measurements of adaptive and non-adaptive Stock-Online services


