An Autonomic Middleware Solution for Coordinating Multiple QoS Controls

Yan Liu[1], Min'an Tan[2], Ian Gorton[3], and Andrew John Clayphan[2]

[1] National ICT Australia, NSW, Australia
jenny.liu@nicta.com.au
[2] University of New South Wales, Australia
{minant,ajc}@cse.unsw.edu.au
[3] Pacific Northwest National Laboratory, U.S.A
ian.gorton@pnnl.gov

Abstract. Adaptive self-managing applications can adapt their behavior through components that monitor the application behavior and provide feedback controls. This paper outlines an autonomic approach to coordinate multiple controls for managing service quality using executable control models. In this approach, controls are modeled as process models. Moreover, controls with cross-cutting concerns are provisioned by a dedicated process model. The flexibility of this approach allows composing new controls from existing control components. The coordination of their dependencies is performed within a unified architecture framework for modeling, deploying and executing these models. We integrate the process modeling and execution techniques into a middleware architecture to deliver such features. To demonstrate the practical utilization of this approach, we employ it to manage fail-over and over-loading controls for a service oriented loan brokering application. The empirical results further validate that this solution is not only sensitive to resolving cross-cutting interests of multiple controls, but also lightweight as it incurs low computational overhead.

1 Introduction

As Service Oriented Architecture (SOA) becomes more widely adopted in large software systems, the typical SOA environment has become more complex. Management of these increasingly complex environments is exacerbated by cross-cutting components and services as well as overlapping SOA environments with service providers beyond the administrator's control. While some human inspection or administration tools can and should be provided, it is unrealistic to expect that all configurations and management can be effectively handled manually. Being fully dependent on manual controls would void the improvements in timeliness and adaptivity gained with an increased level of automation. Consequently, incorporating adaptive and self-managing capabilities into services [4,15] is attracting considerable attention as a means to respond to both the functional and environmental changes that occur after service deployment.

A. Bouguettaya, I. Krueger, and T. Margaria (Eds.): ICSOC 2008, LNCS 5364, pp. 225–240, 2008.

In principle, a system exhibiting adaptive and self-managing capabilities [7,11,12] consists of two parts: (1) a base system that implements the business logic and provides concrete functionalities; and (2) a set of controls that comprise control components for constantly monitoring the system, analyzing the situation and deciding on the actions to affect the system's behavior. When the base system is composed of services in a SOA, the addition of these control components results in adaptive and self-managing service-oriented systems.

In practice, individual control components are dedicated to a specific quality attribute, such as load balancing for performance and scalability or failover for reliability. These are normally constructed independently. In reality, these control components need to be coordinated at runtime to resolve their dependencies that are incurred by cross-cutting concerns. For example, the operation to switch to a backup service may come at a cost of performance by degrading the throughput over a given period of time. While component-based development helps to modularize and encapsulate adaptive and self-managing computation, there is still tight logical coupling and interdependencies between control components. Examples of such tight coupling include systems where the monitoring, analysis and configuration control components explicitly invoke one another without an intervening layer of logical abstraction. Therefore, it is essential to abstract the controls and their dependencies so that their actual implementation is separated from the coordination logic.

In this paper, we propose a novel architecture-based approach to represent, execute and coordinate multiple adaptive and self-managing controls. In this approach, controls are modeled as executable process models. The process engine is integrated with the middleware, allowing the controls to be executed by the same middleware that hosts services. The models can be modified, composed and deployed to the middleware at runtime without affecting the business logic of the services. Moreover, dependencies between controls are also modeled and coordinated using the process models. Our unified approach is realized by an architecture framework, whose default implementation can be further customized and extended to multiple controls. This solution is evaluated by implementing a realistic test scenario. Quantitative measures are collected in terms of the response time overhead, service throughput and CPU usage. The results demonstrate that this architecture solution is lightweight and efficient.

The structure of this paper is as follows: Section 2 discusses the problem through an illustrating example. Section 3 proposes the architectural approach with details of its principle and technical solution. Section 4 further discusses the techniques for coordinating control dependencies. Section 5 presents a case study utilizing this architecture. Section 6 evaluates the overall architecture with measures collected from a test bed. The paper concludes with Section 7.

2 The Problem

We use a typical service oriented system to illustrate the problems involved in coordinating multiple adaptive controls. In addition, we derive the requirement of

the architectural solutions from this example. This application is a representative enterprise integration example derived from industry best practices [6].

Consider the loan brokering application in [6], where a customer submits requests for a loan quote to a loan broker. The loan broker checks the credit worthiness of a customer using a credit agency. The request is then routed to appropriate banks who each give a quote, and the lowest quote is returned to the customer. This application has been deployed (see left of Fig. 1) over an Enterprise Service Bus (ESB) with messaging capabilities provided by Java Messaging Services (JMS), bringing together Web Services, Plain Old Java Objects (POJO) and remote Enterprise Java Beans (EJB). Event flow is driven by the arrival of events. In this application as described in [6], there are two scenarios concerned with adaptive and self-managing controls: (1) failover and (2) overload.

Failover Control. Suppose the responsiveness to requests of the credit agency is in question, and the administrator wants to allow a graceful failover to an alternative credit agency should the primary agency fail. One solution is to insert an additional switching component between the loan broker and the credit agency that can reroute traffic to an alternative credit agency (see Fig 1). In this solution, a test message sensor constantly sends test messages to the credit agency to ensure its correct operation. A notification message is sent to the switch to reroute traffic to a backup credit agency if the test message fails. This forms a feedback control between the test message sensor (feedback) and the switch (control). It is worth noting that this failover occurs at the service level: the primary and secondary services can be from different service providers across the organization boundary. Failures at this level cannot be addressed with system level solutions, such as clustering, and need to be explicitly dealt with at a higher level.

Overload Control. In addition to ensuring the robustness of the credit agency, the administrator also wishes to prevent the loan broker from becoming saturated with requests. As shown in the right of Fig. 1, a throttling component can be used to regulate the flow of requests by limiting the number of concurrent requests being processed. A traffic flow sensor is also used in this situation to detect the flow rate. Beyond the threshold of the system's computing capacity, higher flow rates reduce the number of concurrent processes handling requests and vice versa.

To support ease of service evolution, the composition of control components with existing services should be transparent to business logic. Hence the control logic should not require modification of the original service operation. However, in a component based implementation of the controls, flexibility is reduced as the control logic would be embedded in the components. For example, if the criteria to trigger the failover switch is changed, a rewrite of the basic switching and test message components would be required to coordinate their logic. Another possible solution is to use a "coordinating" component to control the interaction of components in the feedback loop, but again, a change to the hard coded logic is required to this coordinating component if the control structure changes. The high coupling mentioned is still present.

Moreover, introducing control components also creates dependencies between the business and management flows. For example, the loan broker needs to be

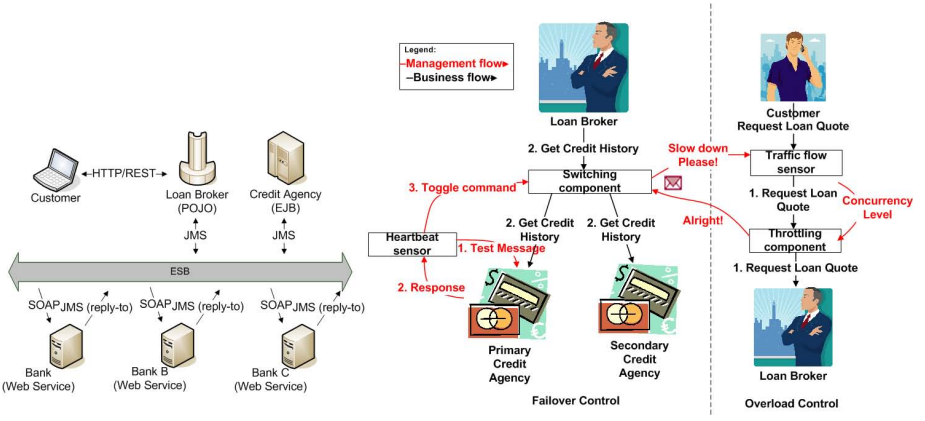


Fig. 1. Self-managing Loan Brokering Service Oriented Systems

aware of the switching component in order to send messages correctly to the switch and not to the credit agency. As more management controls are added, the introduced dependencies both obscure the original business flow as well as reduce the system's flexibility to changes in both flow types.

The desired solution should therefore address the following architectural requirements: (1) represent, execute and coordinate multiple adaptive and self-managing controls; (2) seamless integration of controls, middleware and service business logic; (3) controls can be composed, modified and deployed at runtime; and (4) the solution should be lightweight, otherwise it could adversely degrade overall performance and scalability. We design an architecture framework to address the above issues, since a variety of middleware mechanisms can be leveraged to realize such a framework.

3 The Architecture

We propose a framework to address the relevant architecture issues raised in the prior section. Conceptually, the architecture has five layers. The left of Fig. 2 demonstrates a simplified general architecture with only core components, not specific to any control logic or middleware. The right of Fig. 2 illustrates the customization of the architecture components to specific controls.

The framework aims to provide a modeling based approach towards coordinating multiple controls in service-oriented systems. Adaptive and self-managing controls follow logic that transitions the system from one state into another in response to events dynamically generated at runtime. In addition, the logic represented by the models needs to be executed as well. Given this consideration, we use process models as the tool to present and execute controls. The choice of process models is motivated by their rich semantics, alignment with business goals, integration with Web services and tool support for visual design. The

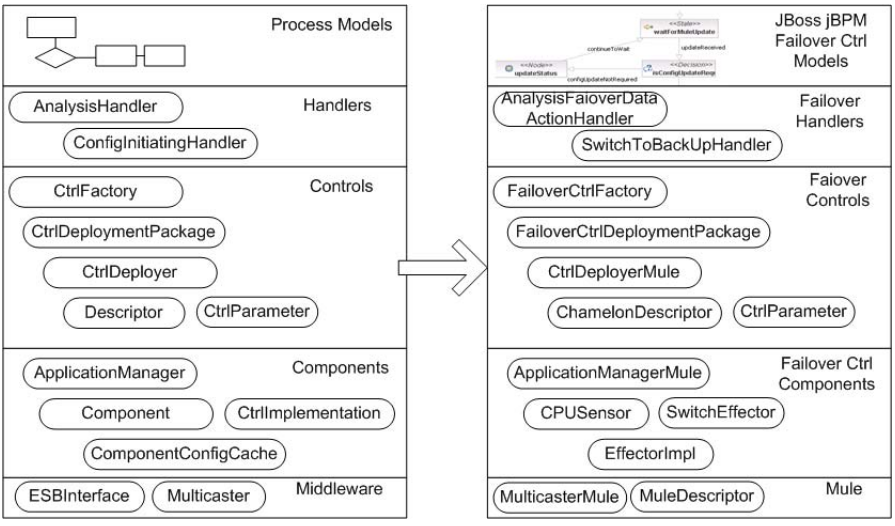


Fig. 2. Simplified Conceptual Architecture

process models can also be executed by process engines with middleware integration features, such as the Oracle BPM engine and JBoss jBPM. In this paper, we use the term *control model* to refer to such a process model designed and executed in a similar way to the JBoss jBPM technology [8].

At the top layer, the control models are firstly designed in diagrams. A model includes nodes for states and transitions triggered by events. Furthermore, these control models are not only for the purposes of presentation, but can be executed. Source code called actions can be attached to either states or transitions of the model. The layer below the control models comprises handlers that encapsulate the action code. Upon entering/leaving a state or before/after triggering a transition, the process engine checks and executes the handler for actions. Our architecture has default implementations for two handlers, *AnalysisHandler* and *ConfigInitialisingHandler*, which are responsible for managing dependencies between control models, and checking a data cache for individual control components respectively. Their usage is addressed in Section 4.1. The combination of these two layers focus on architecture requirement one.

Fig. 3 shows a sample GUI for designing a control model and attaching action code to it. These actions are encapsulated in handlers, and can be executed by a process engine. Such an engine can be embedded at

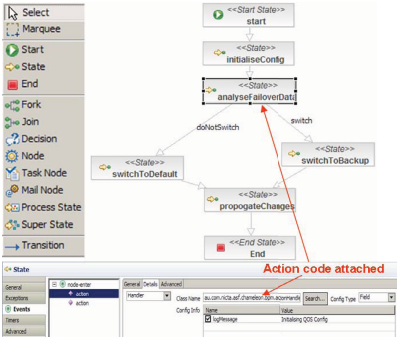


Fig. 3. Control Models and Action Attachment

the middleware level. Therefore, the advantage of using process models in modeling controls is that controls can be visually designed and executed. In addition, the integration of the model execution and the middleware is much simplified by the process engine. This approach is similar to that used in tools such as JBoss jBPM, which is a realization of a full-fledged process engine with IDE support to design process models [8].

The component layer aims to address architecture requirement two. The realization of controls depends on monitoring and actuating components, such as software *sensors* to collect status data to feed into the model, and *effectors* to execute actions. These components are placed into the component layer to separate the control implementation from the business logic. The *ApplicationManager* is responsible for initializing the component instances. As control components participate in service oriented applications, each component has an unique endpoint as its identifier, so that messages can be received from and sent to individual components by service bus middleware.

The control layer aims to fulfill the architecture requirement that controls can be composed, modified and deployed at runtime. Control components are deployed as the unit of the *ControlDeploymentPackage*. Each control has a default *ControlDeploymentPackage* generated by the framework. It contains methods to access all the components involved in a control. Each *ControlDeploymentPackage* uses the *ControlDeployer* to deploy its components. The *ControlDeployer* is responsible for (un)deploying components, creating component *descriptors* and setting the implementation class for each component. This separated deployment of the component instance from its actual implementation further enhances the customization of the adaptive controls. This is because the modification of the implementation does not impact the control models nor the deployment structure, and the implementation can be updated at any time. Once the deployment is finished, an event is broadcasted to other controls about the availability of the new control components.

The bottom layer is the middleware platform. In this paper it is a specific Java ESB – Mule [14]. Mule platform specific mechanisms are used to devise utilities such as concurrency configuration and event multicasting.

In summary, the architecture supports visual and declarative design of adaptive control logic. Controls are modeled as executable process models. These models are executed by a dedicated process engine, which is seamlessly integrated with the middleware. Hence these models can interact with service applications hosted by middleware, receiving and sending messages to realize the control logic. In addition, the architecture supports dynamic update and deployment of controls. As a result, the overall architecture is loosely coupled between business logic and adaptive controls. In the following sections, we further discuss the coordination of multiple controls.

4 Techniques of Coordinating Controls

A challenging issue to solve in this architecture is control dependencies occurring at runtime. Controls designed and deployed independently may involve cross-cutting

concerns. For example, Fig. 1 illustrates that when the failover control takes place, it requires the collaboration from the overload control to slow down its current processing for the period of time that the failover is being executed.

Our architecture can address this issue by the techniques of modeling such concerns as coordination controls. The components coordinated are the *sensors* and *effectors* from individual controls. The dependencies are declared in a control model representing the cross-cutting concern; the specific resolution strategy, be it by heuristic hints or some form of machine learning, consists of implementation specific handlers attached to the process nodes. This leverages on the architecture framework presented, building on the basic idea of sensors, effectors and coordinating components. In the following subsections, we discuss the technical details of achieving such coordination among multiple controls.

4.1 Control Dependencies and Composition

In our approach, the dependencies of controls are declared by developers in a dedicated coordination control, as discussed at the top layer of the architecture. The developer registers controls with dependencies using an *AnalysisHandler* that belongs to the handler layer. This coordination control is modeled and deployed the same way as ordinary QoS controls. When it is deployed, another handler, a *ConfigInitialisingHandler*, checks if an instance of the registered controls exist. After the *ConfigInitialisingHandler* checks the controls and their dependencies, the *AnalysisHandler* can retrieve the *configuration* of individual components in one control. A configuration is part of the control layer. It is an abstraction of what the component does. A configuration contains information about interfaces and properties of a component. Through the configuration, the coordination control and the *AnalysisHandler* can access data that the component contains, invoke its interface on behalf of the coordination and change property values in order to change the control parameters. Sample code is shown in Fig 4.

Using this approach, individual controls are not aware of other controls nor their dependencies. They are transparently managed by coordinating controls. This approach also benefits from the architecture in that a coordination control can flexibly be composed by existing components, which allows quick composition and prototyping of alternative options for adaptive and self-management strategies. An example of composing coordination controls is given in Section 6.1. In addition, coordination controls can be updated, deployed

```
public AnalyseDataHandler(){
            registerDependency("Overload");
            registerDependency("Failover");
} //Register dependencies

OverloadConfig qosConfig = (OverloadConfig)
ConfigInitialisingHandler.getSensorConfig(executionContext,
"Overload");
FailoverConfig failoverConfig = (FailoverConfig)
ConfigInitialisingHandler.getSensorConfig(executionContext,
"Failover");
// retrieve component configuration

ComponentPoolConfig cpConfig = new ComponentPoolConfig();
cpConfig.setId("QosEffector0");
...
if(testeeIsDown(failoverConfig.getTesteeStatus()) &&
      failoverConfig.getMessageBacklog() >
MESSAGE_BACKLOG_THRESHOLD) {
      cpConfig.setInterval(qosConfig.getInterval() +
THROTTLE_INTERVAL);
} else if(cpConfig.getInterval() > 1000){
      cpConfig.setInterval(qosConfig.getInterval() -
THROTTLE_INTERVAL);
}
//set new configuration
```

Fig. 4. Code Sample

or undeployed at runtime. This equips developers with the flexibility to trial-and-test different designs.

4.2 Control Deployment

The deployment of controls takes two steps. First, the control design models in the format of an XML file are deployed to the process engine using an IDE shipped with the process engine. Any action code is attached to the states or transitions in this model. Second, the unit of deployment *ControlDeployment-Package* in our architecture framework is generated, with a mapping to the component implementation record. Following this, the *ControlDeploymentPackage* invokes the *deploy()* method of *ControlDeployer* to deploy itself, creating instances of participating components using their descriptors.

Besides the above functionality of deployment, the architecture requires the ability to intercept incoming requests, and modify outgoing messages. This is also achieved through the control deployment. The control deployment automatically generates intercepting components as a proxy to the intercepted components. The intercepting component takes the identity – the unique endpoint of the intercepted component and forwards requests to and replies from the intercepted components. This feature enables the control composition by redirecting messages to/from any other component transparently to the intercepted components. Fig. 5 depicts components before and after the deployment of the overload control. Details of this control are discussed in Section 5.3.

4.3 Quality Attributes and Optimization

An important architecture requirement discussed in Section 2 is that the computing overhead incurred by this architecture should be optimized. By nature of this service oriented architecture, the optimization problem falls into the category of minimizing messaging overhead. Research on messaging oriented middleware and Web services has demonstrated that the communication rate and payload of the messages have a significant impact on the overall performance and scalability of SOAs [10]. Hence our optimization focuses on reducing the number of messages and their payload with regards to sending collected data among control components including sensors, data analyzers and effectors. Rather than wrapping data as a SOAP attachment, data collected by sensors are stored in a distributed cache. Whenever necessary, a distributed cache is attached with the control components such as software sensors. In this case, we select an open source distributed cache framework – Ehcache [13]. The performance and scalability of Ehcache has proven to satisfy large scale distributed systems [5]. In order to correlate data collected from different sensors, a sensor aggregation component is created at deployment time. In this paper, a default time-based correlation is implemented in the aggregator. The only limitation with using a distributed cache is that the data transition is separated from the web service messages and it is specific to the distributed cache framework.

5 Example Application

We demonstrate our architecture solution using the loan brokering services discussed in Section 2. In addition to verifying the feasibility of our architecture in implementing a practical set of services, we also highlight the flexibility of our architecture for trial-and-test deployments by providing two options to coordinate the failover and overload controls, subsequently referred to as simple and auction-based coordination. In this section, we discuss the specifics of the individual components making up our implementation, as well as two coordination heuristics employed.

5.1 Overload Control

The overload control implements the classic token-bucket algorithm for admission control. It consists of a *Token Bucket Sensor*, a *Throttling Component*, a *Throughput Sensor* and a *Coordination Component*, as shown in Fig 5. The *Token Bucket Sensor* maintains a token bucket with x tokens, where a single token is used for each request. If no tokens are available, the request is dropped and does not enter the system. The token bucket is refilled at rate λ. The *Throttling Component* controls W, the number of concurrent requests that can be processed. Each processed request is delayed by a throttling interval I. The *Throughput Sensor* measures δ, the rate of requests being processed by the system. Finally, the *Coordination Component* constantly aggregates the throughput δ and the number of tokens left in the token bucket. It then feeds the status to the control model in the process engine, and multicasts to effectors the decision on the new values of λ, W and I accordingly. The adjustment of λ is given by:

$$\alpha * \frac{W}{I} + (1 - \alpha) * \delta$$

where α is a tuning parameter to adjust the component weight.

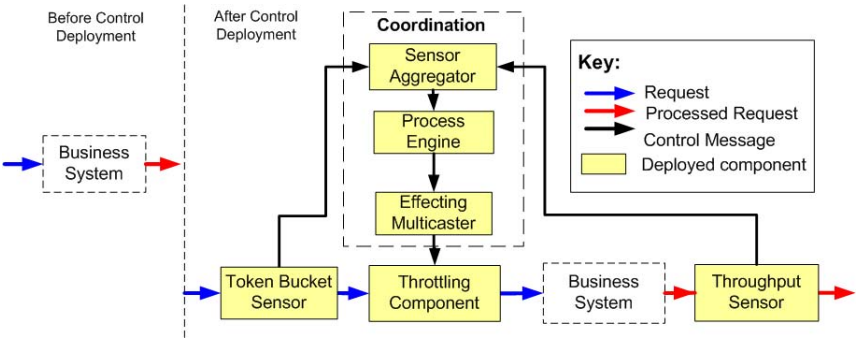


Fig. 5. Overload Control Deployment

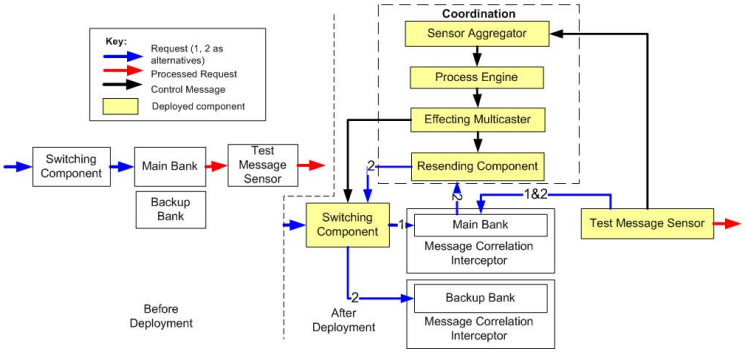


Fig. 6. Failover Control Deployment (switch in *off*[1] or *on*[2] mode)

5.2 Failover Control

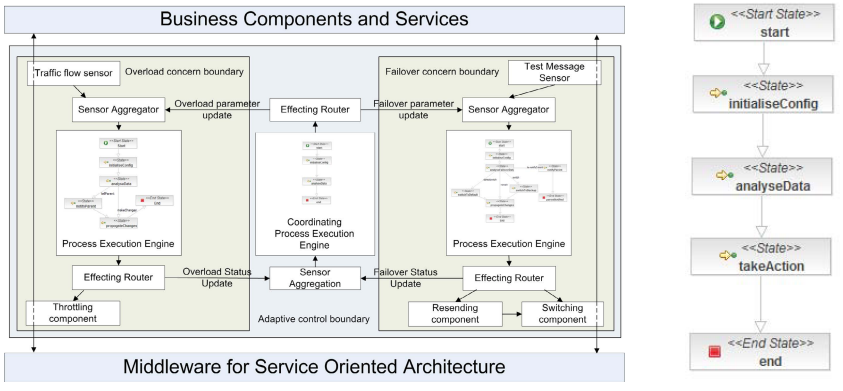
The failover control shown in Fig. 6 consists of a *Test Message Sensor*, a *Switching Component*, a *Resending Component* and a *Coordination Component*. The *Test Message Sensor* constantly sends test messages to the main service. It uses the test messages to determine if the main service is active or has failed. The *Coordination Component* constantly receives inputs from the *Test Message Sensor* and adjusts the state of the *Switching Component* (on or off). If the main service has failed, the *Switching Component* routes incoming requests to the active service when its state is toggled to on by the *Coordination Component*. A *Message Correlation Interceptor* maintains a queue of messages by intercepting incoming requests to the main service. When a request is successfully routed, the request is removed from the queue. The *Resending Component* sends unprocessed requests from the *Message Correlation Interceptor* to the active services when the *Switching Component* is toggled.

5.3 Coordinating Multiple Controls

To coordinate these controls, our general approach is to let the overload control throttle the workload when failover takes place. In our implementation, two options are provided to realise this approach.

Our implementation of the architecture is deployed as shown in Fig. 7(a), which also shows the failover and overload controls employed. The core of the coordination is the control model shown in Fig. 7(b). This was created using the JBoss jBPM process model designer. As both *Coordination Components* multicast their status data, the coordination between these two controls collects updated status data from each control using its *SensorAggregator* and sends out action decisions through the *EffectingRouter*. Handlers are attached to nodes and transitions to realize the two control options: (1) simple coordination and (2) auction-based coordination.

The simple coordination control tunes the concurrency level of processing new, incoming requests in the middleware. The tuning is based on the number



(a) (b)

Fig. 7. Implemented Example Architecture (a) and Coordination Control Model (b)

of messages yet to be resent by the failover control. This control is easier to implement, but has limitations when producing the optimal concurrency levels for a large set of services.

In the auction-based control, requests being resent by the failover control and new incoming requests at the overload control bid for tokens. Tokens are dynamically allocated to requests both from failover and overload controls. Only requests with a token can be processed, otherwise there is a wait for the next available token. In general, the auction-based control incurs more overhead in communication as a bid is multicast. However, the auction-based control is more practical and suitable when it is nontrivial to tune the concurrency level of the middleware.

Both options reuse the failover and overload controls, and it should be noted that the control model is identical for both options. The difference is in the way each of them process status data and the actions taken. This is reflected by the different options having different handlers attached to the appropriate control model nodes.

5.4 Discussions

In this example, process modeling tools and middleware mechanisms are used to customize the general architecture to a specific implementation. As mechanisms from middleware (such as interceptors) and modeling features from the process engine (such as handlers) are commonly supported, other process modeling tools and service bus middleware can be applied to the framework. The only condition however, is that the process engine should be able to communicate with the middleware. For example, Mule provides a common interface for process engines to access its features [14]. We could have used an Oracle BPM implementation of the interface instead of JBoss jBPM, without any change to other implemented components. This illustrates the generic nature of our architectural solution.

6 The Evaluation

Each option of the coordination control is measured and the results are compared to identify key performance factors.

6.1 Testbed Setup

We deploy the loan brokering services as shown in Fig. 1 on the Mule ESB. The credit agencies are developed as Java EJBs and deployed on a JBoss application server. Bank services are Web services deployed on Apache Tomcat servers. The brokering service is a Mule application, and it communicates with other services through Mule. The adaptive controls (failover and overload) are designed using JBoss jBPM and their models are deployed into the jBPM engine. The handlers and control components are built upon the architecture framework discussed in Section 3 using Java. Together with the process engine, the models and components are deployed on Mule.

We also develop a simple workload generator which injects a number of requests into the system under test with a bounded random time between request arrivals. For example, the interval [75,200] means the request arrival time bound is between 75 to 200 milliseconds. In order to observe performance, a simple console showing charts of metrics was developed (see Fig. 8 for example).

The testing environment includes two identical Windows XP machines with 2.4GHz Dual Xeon Processors, one hosting loan broker services, credit agencies and adaptive controls and the other hosting five bank services which are identical to simplify implementation. The workload generation are 500 requests with the interval [75,225]. If the overload control is enabled, the throttling component controls W, the number of concurrent requests that can be processed. W is set to 100 initially.

6.2 Performance Results

We test four scenarios: (1) only the overload control is enabled; (2) only the failover control is enabled; (3) simple coordination; and (4) auction-based

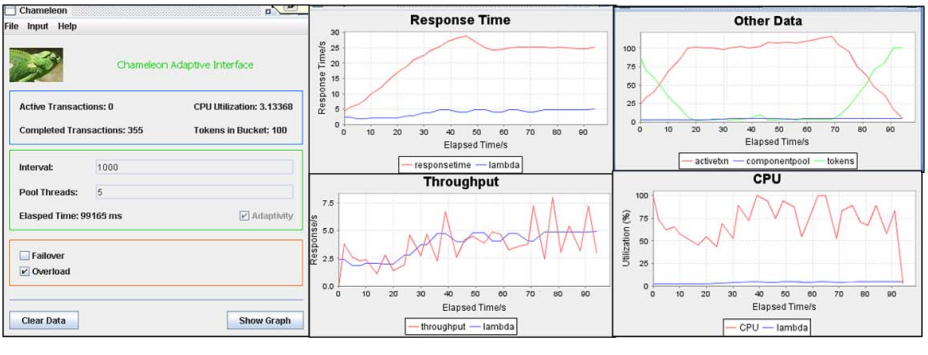


Fig. 8. Overload Control Only Observation

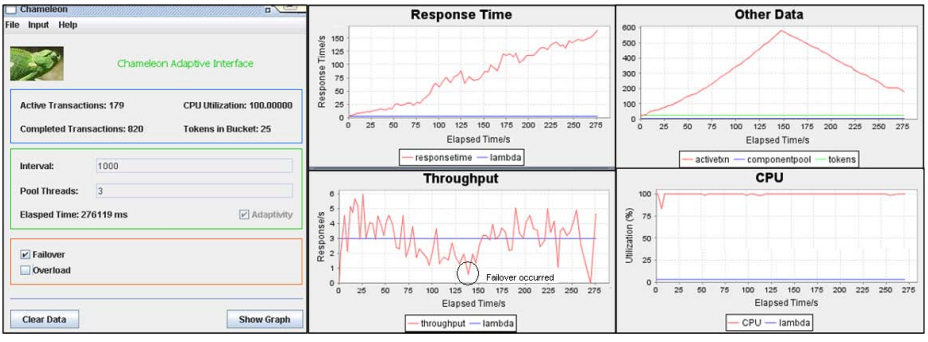


Fig. 9. Failover Control Only Observation

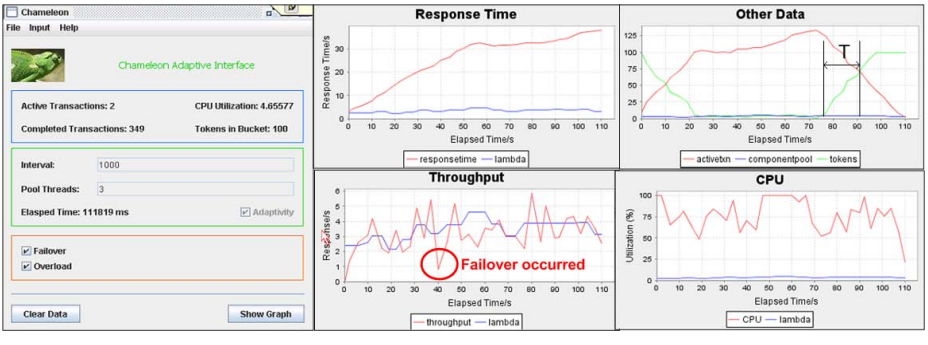


Fig. 10. Simple Coordination Observation

coordination. Obviously in (3) and (4) both failover and overload controls are enabled. The same environment configurations are used for each test. Fig. 8 to Fig. 11 show sample performance measurements from the testing scenarios.

Fig. 8 shows that the overload control is efficient in self-management of performance and scalability. It is shown on the other data chart (top right of Fig. 8) that approximately after 20s has elapsed, the token number hits zero, meaning there are already 100 requests being processed. From the CPU chart, it shows that the CPU utilization starts increasing and it triggers the overload control before 40s elapsed time. The response time chart illustrates that the overload control takes effect around 45s elapsed time, and the response time reaches a plateau rather than continuing to linearly increase.

Fig. 9 shows the failover control also works. At around elapsed time 140s, the primary credit agency service is deliberately shut down, and the requests are routed by the failover control to the secondary credit agency service. This is consistent with the other data chart that shows the active transactions reach the peak at around elapsed time 140s, and then degrades when the failover occurs. The CPU resource is saturated without the overloading control and the response time increases. These separated performance testing scenarios confirm

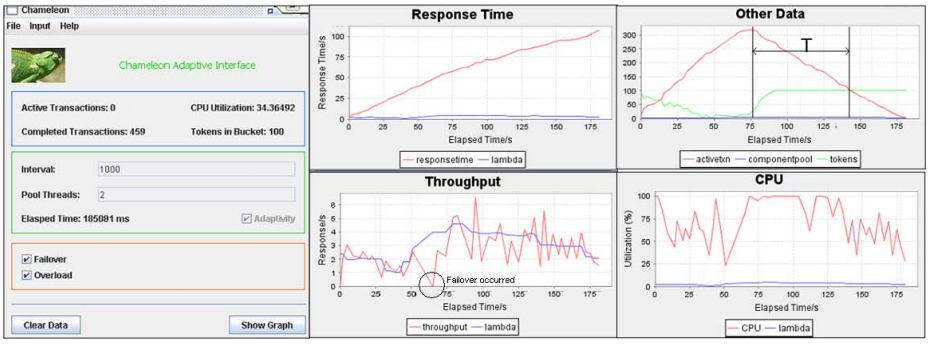


Fig. 11. Auction-based Coordination Observation

the motivation for coordinating two controls to yield a better quality of service. The results from a single control indicate that the overhead of the architecture framework itself is insiginificant, and the performance factors are determined by the adaptive self-managing strategies.

The results of the simple coordination are shown in Fig. 10. Compared with the case of failover control only, the coordination helps to improve the performance. Now the response times reach the plateau and the CPU utilization is not saturated. From the results of the auction-based coordination shown in Fig. 11, the performance improvement is less than the simple coordination, which we attribute to the additional communication overhead incurred in the auction-based coordination as mentioned in Section 5.3. An interesting observation is the time (annotated as **T** in the diagrams) spent on processing queued requests. Requests are put in a queue by the overload control when all the tokens are consumed, and are only processed when the token bucket is refilled and more tokens are available. The auction-based coordination spent longer time (**T**) than the simple option, which contributes to the degradation of performance.

It is worth noting that our focus is not on studying and evaluating individual coordination controls but rather on demonstrating the practical usage of the architecture to compose them. The difference observed by prototyping and testing shows that our architectural solution can be applied to the development of realistic self-managing service oriented systems. The resulting architecture framework to support this solution is useful to trial different control options.

7 Related Work

Applying business process models to support self-managing software systems has recently been investigated in various domains [2,18]. For example, Verma and Sheth envisioned autonomic web processes [18]. Similar to the core of this paper, they elevated autonomic computing concepts from infrastructure to a process level. Their paper discussed existing technologies and steps needed to shorten the gap from current process management to autonomic web processes.

In this paper, we present a practical architecture solution to integrate process management with middleware-based services.

Extensive research has been done in the translation of business process models to execution languages, and there has been an increasing adoption of business process modeling (BPM) tools to coordinate business process flows, as opposed to hard-coded application logic. A good summary of the relevant techniques and tools is covered in [1]. As the core of our approach, control models leverage business process models to represent adaptive logic. It is an open research question with regards to the integration of the semantics essential to adaptive self-management with general business process modeling capabilities. The on-going research in evaluating the expressiveness of business process definitions will contribute insights to this research space [19].

A comprehensive survey [3] discussed existing technologies that could enable dynamic composition of adaptive software. It also classified different approaches by how, when and where composition might 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.

8 Conclusion

This paper proposes an approach to develop adaptive self-managing controls for service oriented systems. The contribution of this work is twofold. Firstly, it leverages process models to design adaptive controls with a visual context. Moreover, an architecture framework is built to integrate the models with a middleware platform and enable the execution of the design models. Secondly, multiple controls can be coordinated using this solution. Different options can quickly be prototyped by composing the coordination from existing control components. This architecture-based solution provides benefits towards modifiability, reusability and maintainability of self-managing service oriented systems. The quantitative evaluation is based on a realistic enterprise service bus application. The results demonstrate the performance efficiency of this approach. Our on-going work involves developing tools to simulate the controls when it is designed and deployed from the process models. The simulation tools combined with the rest of the architecture framework further help developers test their adaptive and self-managing controls at an early design stage.

References

1. van der Aalst, W.M.: Business process management demystified: A tutorial on models, systems and standards for workflow management. In: Lectures on Concurrency and Petri Nets, pp. 1–65 (2004)
2. Baresi, L., Guinea, S., Pasquale, L.: Self-healing bpel processes with dynamo and the jboss rule engine. In: ESSPE 2007: International workshop on Engineering of software services for pervasive environments, pp. 11–20. ACM, New York (2007)

3. McKinley, P.K., Sadjadi, S.M., Kasten, E.P., Cheng, B.H.C.: Composing adaptive software. Computer 37(7), 56–64 (2004)
4. Naccache, H., Gannod, G.C.: A self-healing framework for web services. Icws 00, 345–398 (2007)
5. Gorton, I., Wynne, A., Almquist, J., Chatterton, J.: The MeDICi Integration Framework: A Platform for High Performance Data Streaming Applications. In: WICSA 2008: 7th Working IEEE/IFIP Conference on Software Architecture, pp. 95–104. IEEE Computer Society, Los Alamitos (2008)
6. Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Professional, Reading (2003)
7. IBM. An architectural blueprint for autonomic computing. IBM Autonomic Computing (2004)
8. JBoss jBPM, `http://www.jboss.com/products/jbpm`
9. Foster, H., Uchitel, S., Magee, J., Kramer, J.: Tool Support for Model-Based Engineering of Web Service Compositions. In: Proc. of Intl. Conf. on Web Services (ICWS 2005), pp. 95–102. IEEE Computer Society, Los Alamitos (2005)
10. Juse, K., Kounev, S., Buchmann, A.: PetStore-WS Measuring the Performance Implications of Web Services. In: CMG 2003: Proc. of the 29th International Conference of the Computer Measurement Group (2003)
11. Kephart, J.O.: Research challenges of autonomic computing. In: ICSE 2005: Proceedings of the 27th international conference on Software engineering, pp. 15–22. ACM, New York (2005)
12. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: FOSE 2007: 2007 Future of Software Engineering, pp. 259–268. IEEE Computer Society, Los Alamitos (2007)
13. Luck, G., Suravarapu, S., King, G., Talevi, M.: EHCache Distributed Cache System, `http://ehcache.sourceforge.net/`
14. Mule ESB, `http://mule.mulesource.org/`
15. P.M., et al.: The wsdm of autonomic computing: Experiences in implementing autonomic web services. In: SEAMS 2007: Proceedings of the 2007 International Workshop on Software Engineering for Adaptive and Self-Managing Systems, p. 9. IEEE Computer Society, Los Alamitos (2007)
16. Anthony, R.J.: Policy-based techniques for self-managing parallel applications. Knowl. Eng. Rev. 21(3), 205–219 (2006)
17. Kumar, V., Cooper, B.F., Eisenhauer, G., Schwan, K.: Enabling policy-driven self-management for enterprise-scale systems. In: HotAC II: Hot Topics in Autonomic Computing on Hot Topics in Autonomic Computing, pp. 4–23. USENIX Association (2007)
18. Verma, K., Sheth, A.P.: Autonomic Web Processes. LNCS. Springer, Heidelberg (2005)
19. Zhu, L., Osterweil, L., Staples, M., Kannengiesser, U., Simidchieva, B.: Desiderata for languages to be used in the definition of reference business processes. International Journal of Software and Informatics 1, 37–65 (2007)