Transparent Autonomization in Aggregate Web Services Using Dynamic Proxies

Technical Report FIU-SCIS-2006-02-01

February 2006

Onyeka Ezenwoye and S. Masoud Sadjadi

Autonomic Computing Research Laboratory (ACRL)
School of Computing and Information Sciences
Florida International University
11200 SW 8th St., Miami, FL 33199
{oezen001,sadjadi}@cs.fiu.edu

Abstract

We recently introduced RobustBPEL [13], a software toolkit that provides a *systematic* approach to making existing aggregate Web services more tolerant to the failure of their constituent Web services. Using RobustBPEL, we demonstrated how an aggregate Web service, defined as a BPEL process, can be instrumented automatically to monitor its partner Web services at runtime and replace failed services via a generated proxy. While in the previous work the proxy is *statically* bound to a limited number of alternative Web services, in this paper we propose an extension to the RobustBPEL toolkit to generate a proxy that *dynamically* discovers and binds to existing services. Further, we present details of the generation process, the architecture of the dynamic proxy, and finally use a case study to demonstrate how the generated dynamic proxy is used to support self-healing and self-optimization (specifically, to improve the fault-tolerance and performace) in an instrumented BPEL process.

Keywords: Web service monitoring, BPEL processes, self-healing, self-optimization, dynamic service discovery.

1 Introduction

Web services are facilitating the uptake of Service-Oriented Architecture (SOA) [7], allowing business organizations to electronically interact with one another over the Internet. In this architecture, reusable, self-contained and remotely accessible application components, which are exposed as Web services, can be integrated to create more course-grained aggregate services (*e.g.*, a flight reservation service). For this, high-level workflow languages such as BPEL [27, 28] can be used to define aggregate services (*a.k.a.*, business processes) that constitute a number of related services (*a.k.a.*, business functions) [17]. Unfortunately, these types of business processes are known to be very fragile. According to [21], about 80 percent of the total amount of time used in developing business processes is spent in exception management.

The integration of multiple services, which are potentially developed and maintained on heterogeneous environments, introduces new levels of complexity in management. The management of aggregate services

goes beyond the administration boundaries of individual services, involving different policies unknown to the aggregate services. Also, services interacting with these aggregate services are often geographically scattered and communicate via the Internet, which is unreliable and prone to failure. Given the unreliability of such communication channels, the unbounded communication delays, and the autonomy of the interacting services, it is difficult for developers of business processes to anticipate and account for all the dynamics of such interactions. In addition, the high-availability nature of some business processes requires them to work in the face of failure of their constituent parts [5, 11]. It is then important to make aggregate services more resilient to the failure of their partner services.

Autonomic computing [16] promises to solve the management problem by embedding the management of complex systems inside the systems themselves, freeing the users from potentially overwhelming details. A Web service is said to be autonomic if it encapsulates some autonomic attributes [15]. Autonomic attributes include self-configuration, self-optimization, self-healing, and self-protection [16]. The focus of our ongoing research is to encapsulate *self-healing* behavior in business processes in order to make them more resilient to the potential failures of their partner services. Specifically, we aim to make an aggregate Web service continue its valid function after one or more of its constituent Web services have failed.

We recently introduced RobustBPEL [13], a software toolkit that provides a *systematic* approach to making existing aggregate Web services more tolerant to the failure of their constituent Web services. Using RobustBPEL, we demonstrated how an aggregate Web service, defined as a BPEL process, can be instrumented automatically to monitor its partner Web services at runtime to check if these services actually fulfill their service contracts. To achieve this, events such as faults and timeouts are monitored from within the adapted process. We showed how our adapted process is augmented with a *static* proxy that replaces failed services with predefined alternatives.

While in the previous work the proxy is *statically* bound to a limited number of alternative Web services, in this paper we propose an extension to the RobustBPEL toolkit to generate a proxy that *dynamically* discovers and binds to existing services. The recent proliferation of Web services has convinced us that more appropriate services may become available after the composition and deployment of the BPEL process and its corresponding static proxy. So, it makes sense that upon failure or delay of any of the partner Web services of the BPEL process, an equivalent service can be discovered dynamically (at run-time) to serve as a substitute for the service. In doing this, we improve the fault tolerance and performance of BPEL processes by transparently adapting their behavior. By *transparent* we mean that the adaptation preserves the original behavior of the business process and does not tangle the code that provides self-healing and self-optimization behavior with that of the business process [26]. This transparency is achieved by using a *dynamic* proxy that encapsulates the autonomic behavior (adaptive code).

In this paper, we show how new components can be *dynamically* discovered and introduced to BPEL processes in order to support self-healing behavior. We show the design and implementation of the dynamic proxy that is used to achieve this goal and use a case study to demonstrate the autonomic behavior of the dynamic proxy. The rest of this paper is is structured as follows. Section 2 provides a background on our architecture. Section 3 describes the dynamic proxy and how it is generated. In section 4 we use a case study to demonstrate our approach. Section 5 contains some related work. Finally, a conclusion and discussion on further research is in Section 6.

2 Architectural Background

A Web service is a software component that can be programmatically accessed over the Internet. The interface to the functionality provided by a Web service is described in Web services Description Language (WSDL) [9]. To make a call on these functions remotely, a messaging protocol such as SOAP [14] can be used. The goal of the Web services architecture [7] is to simplify application-to-application integration.

The technologies in Web services are specifically designed to address the problems faced by traditional middleware technologies in the flexible integration of heterogeneous applications over the Internet. Its lightweight model has neither the object model nor programming language restrictions imposed by other traditional middleware systems (*e.g.*, DCOM and CORBA) and its messaging protocol ensures that packets are able to traverse Internet firewalls.

Applications that provide specific business functions (*e.g.*, price quotation) are increasingly being exposed as Web services. These services then become reusable components that can be the building blocks for more complex aggregate services (business processes). To facilitate the creation of these business processes, a high-level workflow language, such as Business Process Execution Language (BPEL) [2], is often used. BPEL provides many constructs for the management of the process including loops, conditional branching, fault handling and event handling (such as timeouts). To make a BPEL process fault tolerant, BPEL fault handling activities, such as `catch` or `catchAll` constructs, can be used. However, we aim to separate the task of making a BPEL process more robust from the task of composing the business logic of the process [13].

RobustBPEL is a toolkit developed as part of the *transparent shaping* paradigm. Transparent shaping provides transparent application adaptation by transparently augmenting existing applications with *hooks* that intercept and redirect interaction to *adaptive code*. The adaptation is transparent because it preserves the original behavior and does not tangle the code that provides the new behavior (adaptive code) with the application code [26]. By adapting *existing* applications, transparent shaping aims to achieve a separation of concerns [12, 18]. That is, enabling the separate development the applications functional requirements (the business logic) from the non-functional requirements. By non-functional we mean those program attributes that do not provide a given business value (*e.g.*,fault tolerance). We use transparent shaping [24] to create *adapt-ready* versions of existing BPEL processes. A program is known as *adapt-ready* if its behavior can be changed with respect to the changes in its environment or its requirements. The goal is to get the adapt-ready BPEL to exhibit the desired autonomic behavior.

The first step of transparent shaping is to weave generic hooks at *sensitive joinpoints* in the target program. These joinpoints are certain points in the execution path of the program at which adaptive code can be introduced at run time. Key to identifying joinpoints is knowing where in the application at which monitoring is required and inserting appropriate code (hooks) to do so. Because a BPEL process is an aggregation of services (applications), the most appropriate place to insert interception hooks is at the *interaction joinpoints* [25]. The monitoring code we insert is in the form of standard BPEL constructs to ensure the portability of the modified process.

We adapt the existing BPEL process by identifying points in the process at which external Web services are invoked (interaction joinpoints) and then wrapping each of those invocations with a BPEL `scope` that contains the desired fault and event handlers. A fault can be a programmatic error generated by a Web service partner of the BPEL process or unexpected errors (*e.g.*, service unavailability) from the Web service infrastructure. The following XML code (Figure 1) is an example of a service invocation in BPEL. Lines 3 and 4 identify the interface (`portType`) of the partner and what method (`operation`) the invocation wishes to call.

The invocation showed in Figure 1 is identified and wrapped with monitoring code. The code in Figure 2 shows how the invocation looks like after the monitoring code is wrapped around it. The unmonitored invocation is first wrapped in a `scope` container which contains fault and event handlers (lines 2-11 and 12-21 respectively in Figure 2). A `catchAll` fault handler is added (lines 3-10) to the `faultHandlers` to handle any faults generated as a result of the invocation of the partner Web service. The fault-handling activity defined is the invocation of the proxy Web service (lines 4-9). When a fault is generated by the partner service invocation, this fault is caught by the `catchAll` and the proxy service is invoked to substitute for the unavailable or failed service.

A similar construct is used for the event handler. An `onAlarm` event handler (lines 13-20) is used to

```
1. <invoke name="invokeApprover"
2.         partnerLink="approver"
3.         portType="loanApprovalPT"
4.         operation="approve"
5.         inputVariable="request"
6.         outputVariable="approvalInfo">
7. </invoke>
```

Figure 1: An unmonitored invocation.

specify a timeout. An `onAlarm` clause is used to specify a timeout "event" in BPEL. A timeout can be used, for instance, to limit the amount of time that a process can wait for a reply from an invoked Web service. That is, a duration within which the partner service invocation must complete. If the partner service fails to reply within the stipulated time, the proxy service is invoked (lines 14-19) as a substitute. The duration of the timeout (line 13), can vary depending on user preference or application.

```
1. <scope>
2.   <faultHandlers>
3.     <catchAll>
4.       <invoke name="InvokeProxy"
5.               partnerLink="proxy"
6.               portType="proxyPT"
7.               operation="approve"
8.               inputVariable="request"
9.               outputVariable="approvalInfo"/>
10.    </catchAll>
11.  </faultHandlers>
12.  <eventHandlers>
13.    <onAlarm for="'PT15S'">
14.      <invoke name="InvokeProxy"
15.              partnerLink="proxy"
16.              portType="proxyPT"
17.              operation="approve"
18.              inputVariable="request"
19.              outputVariable="approvalInfo"/>
20.    </onAlarm>
21.  </eventHandlers>
22.  <invoke name="invokeApprover"
23.         partnerLink="approver"
24.         portType="loanApprovalPT"
25.         operation="approve"
26.         inputVariable="request"
27.         outputVariable="approvalInfo">
28.  </invoke>
29.</scope>
```

Figure 2: A monitored invocation.

The job of the proxy Web service is to discover and bind *equivalent* Web services that can substitute for the monitored services. By replacing failed and delayed services with their equivalents (substitutes), the proxy Web service provides self-healing and self-optimization autonomic behavior to the BPEL process,

thereby making the BPEL process autonomic. The interface for the generated proxy Web service is exactly the same as that of the monitored Web service. This is why we call this proxy *specific*. Thus, the operations and input/output variables of the proxy are the same as that of the monitored invocation. When more than one service is monitored within a BPEL process, the interface for the specific proxy is an aggregation of all the interfaces of the monitored Web services.

Although the adapt-ready BPEL process remains a functional Web service and the proxy is an autonomic Web service (encapsulates autonomic attributes), functional Web services can behave in an autonomic manner by using autonomic Web services [15]. At this point in our research, we make the assumption that two services are *equivalent*, if they implement the same port type (and thus business logic). A port type is similar to an interface in the Java programming language. So, when two Web services implement the same port type, only their internal implementations may vary, their interfaces remain the same. In other words, applications with the same functional requirement are equivalent, regardless of implementation. At runtime, if monitored service fails (or an invocation timeout occurs), the input message for that service is used as input message for the proxy. The proxy invokes the equivalent service with that same input message. A reply from the substitute service is sent back to the Loan Approval BPEL process via the proxy.

We developed a generator to automatically generate the adapt-ready version of a given BPEL process and its associated static proxy service. The generator (Figure 3), needs as input three sets of documents: (1) the original BPEL process, (2) the template from which the proxy Java class is generated, and (3) the WSDL descriptions of all the substitutes for the monitored services. The WSDL files for the substitutes are needed so that binding stubs for the substitute services. These stubs are then statically associated with the generated proxy class.

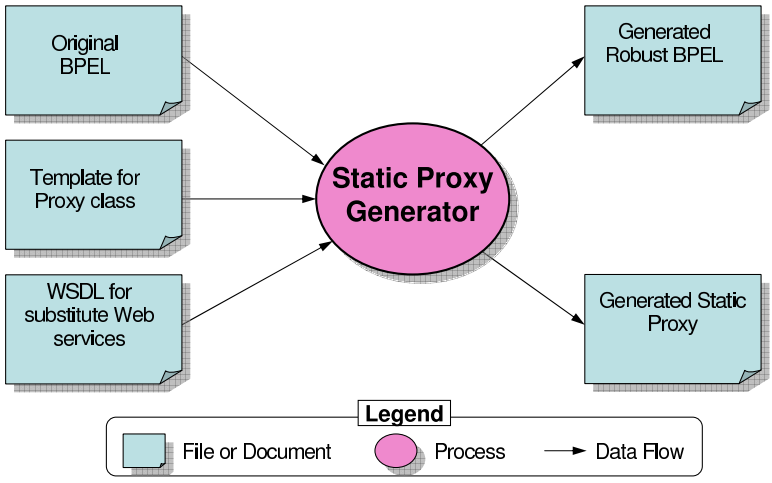


Figure 3: Inputs and outputs of the static proxy generator.

3 Dynamic Proxy

The main difference between the static and the dynamic proxies is that when static discovery is used, services that can substitute for the monitored services are noted and tightly associated with the code for the static proxy. For the dynamic proxy, a look-up mechanism is utilized to query a service registry at runtime for services that can be used to replace failed services. The registry technology used in this case is the Universal Description, Discovery and Integration protocol (UDDI) [3], which is a specification for the publication and discovery of Web services. UDDI specifies a set of data structures, messages and API for creating and maintaining information about Web services in distributed registries. UDDI utilizes existing Web service

technologies, such as HTTP, XML, and SOAP.

Figure 4 illustrates how UDDI fits within the Web services stack. The registry allows for three categories of information to be published: (1) *white pages* that contain contact information such as the name, address, telephone number of a given business; (2) *yellow pages* that contain information that categorizes businesses based on some existing taxonomies; and (3) *green pages* that contain technical information about the Web services provided by the published businesses (this can include the URL of the service and its WSDL). A provider can publish descriptions of its services with the registry. A requester can send inquiries for services in the form of SOAP messages to the registry, which then replies with information about services.

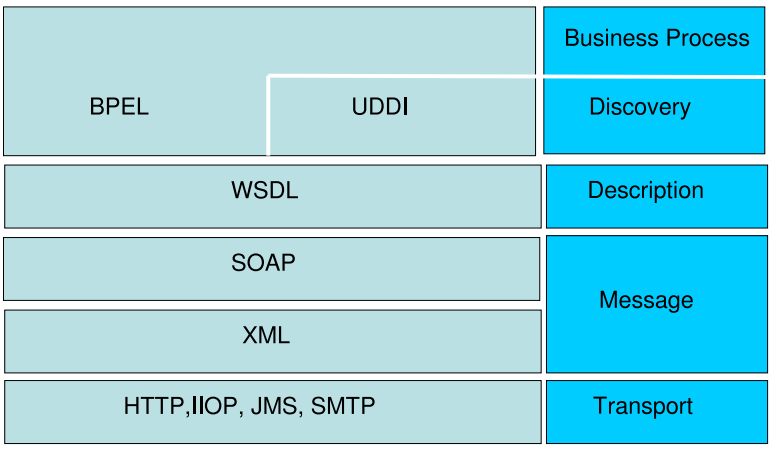


Figure 4: The Web services stack.

For this phase of our work, we have developed a method for generating dynamic proxies to be associated with specific BPEL processes. When the dynamic proxy is invoked upon failure of a monitored service, the proxy makes queries against the yellow pages of a UDDI registry for services in the same classification. The classification used in this case is based on the service type. The result of this query (if any services are found) is used to query the green pages for their binding information. These queries are carried out with the use of the UDDI API. At this stage of our work, no selection criteria is used when multiple services are found. The services are just chosen arbitrarily from a list.

The generator for the dynamic proxy, as illustrated in Figure 5, differs slightly from that of the static proxy in that it needs as input four sets of documents, rather than three. They are: (1) the original BPEL process, (2) a template for the proxy Java class, (3) a generic binding stub for substitute Web services, and (4) the stubs that contain the bit of code necessary for queries to be made to the UDDI registry. The generic binding stub is needed in order for the proxy to be able to bind to any discovered substitutes. The binding stub is generic because at this point we assume that services are equivalent if they implement the same port type. When a substitute service is discovered from the registry, the generic stub is updated with information about the physical location of that service. The proxy can then invoke the substitute using the generic stub. The output of the generator is the monitored BPEL and a proxy Web service that utilizes dynamic discovery.

4 Case Study

In this section, we use a case study to demonstrate the self-healing and self-optimization behavior of our generated dynamic proxy. We start by describing the case application, then we present the configuration of the experiment environment. Finally we, show the results of the experiment.

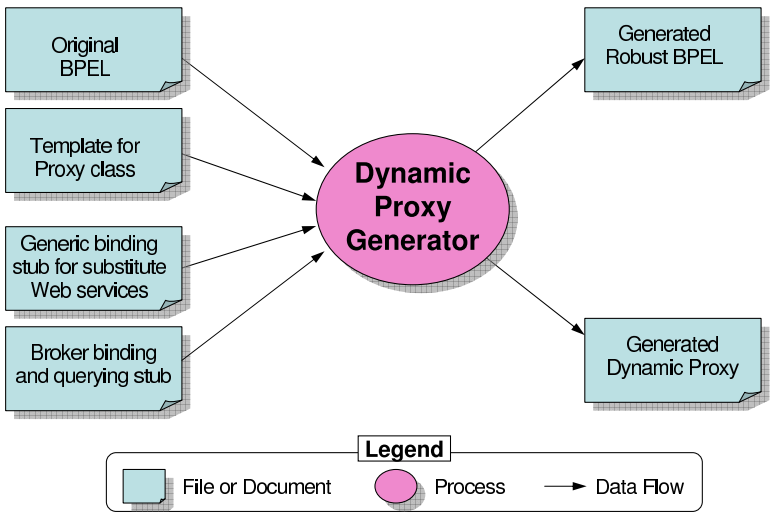


Figure 5: Inputs and outputs of the dynamic proxy generator.

4.1 The Example Application

The example application we use is a loan-approval process that is commonly used as a sample BPEL process [1]. The loan-approval BPEL process is an aggregate Web service (`LoanApproval`) composed of two other Web services: a risk assessor service (`LoanAssessor`) and a loan approver service (`LoanApprover`). The loan-approval process implements a business process that uses its two partner services (`LoanAssessor` and `LoanApprover`) to decide whether a given individual qualifies for a given loan amount. Note: in this case study, we assume that we have access to the loan-approval BPEL process, but the two other Web service partners are developed, deployed and managed by third parties (outside our control).

As illustrated in Figure 6, the loan-approval BPEL process receives as input a loan request (`ReceiveCustomerRequest`). The loan request message comprises two variables: the name of the customer and the loan amount (not shown in the figure). If the loan amount is less than $10,000, then the risk assessor Web service is invoked (`InvokeRiskAssessor`), otherwise the loan approver Web service is invoked (`InvokeLoanApprover`). The risk assessor and the loan approver services take as input the loan request message (not shown in the figure). After the risk assessor is invoked, the BPEL process expects to receive as reply a risk assessment message.

This risk assessment message is a string with a value of either “high” or “low”. When the risk assessment is “low”, it means that the loan is approved and the loan-approval process sends an approval message (with “yes” value, `AssignYestoAccept`) to the customer and terminates (`ReplyCustomerRequest`). If the risk assessment message is “high”, the loan approver service is invoked (`InvokeLoanApprover`). The loan approver service returns a loan-approval message (either “yes” or “no”), which is then sent as reply to the customer (`AcceptMessageToCustomer`). Both the risk assessor service and the loan approver service can also return a predefined fault message to the BPEL process. When any of these services reply with a fault message, the BPEL process sends an error message to the user and terminates (not shown in the figure).

We note that both the risk assessor and the loan approver Web services were implemented previously by ActiveWebflow in Java [1].

4.2 The Experiment Configuration

We have used a total of five machines to conduct our experiments: PC-1, PC-2, PC-3, PC-4 and PC-5. All the machines have Windows XP as their operating system and running on each is an Apache Tomcat

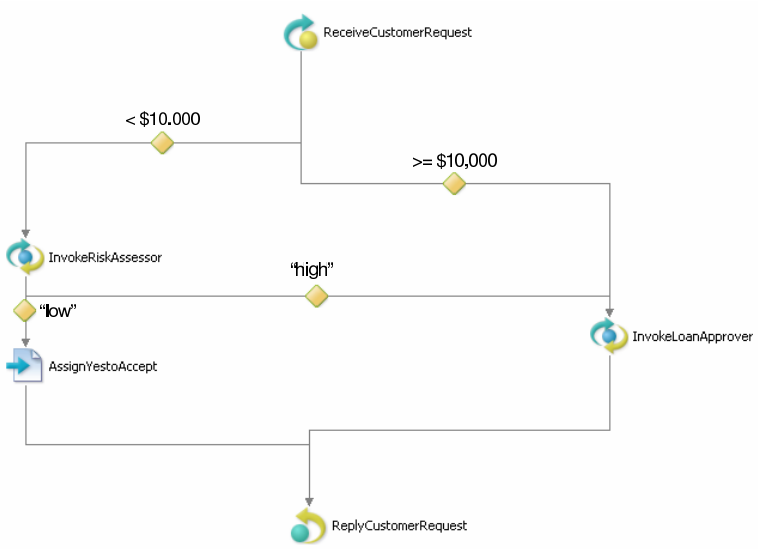


Figure 6: The work-flow diagram of the loan-approval BPEL Process.

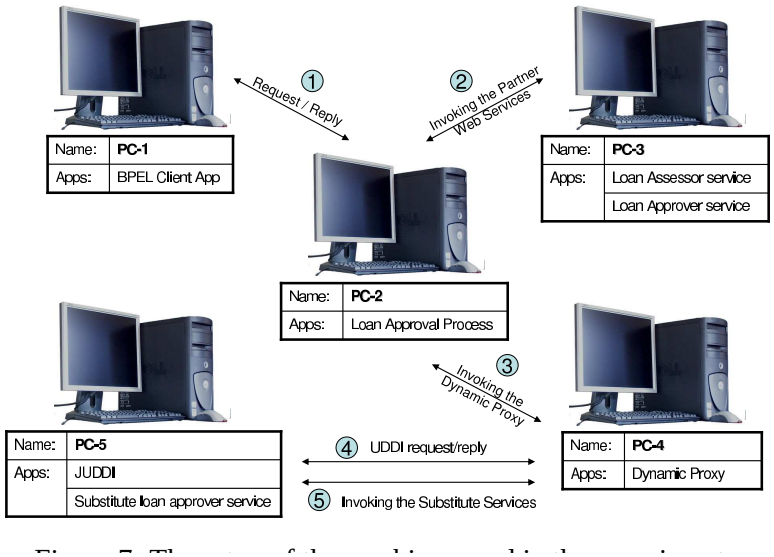
web server. Also deployed on each web server is an Apache AXIS SOAP engine. A multi-threaded BPEL client application written in Java is deployed on PC-1. An Active-Bpel engine [1] is deployed on PC-2. This BPEL engine is deployed as a web application under the Tomcat web server and is the platform on which the loan-approval BPEL process was hosted. The loan-assessor and loan-approver Web services are deployed on PC-3. The dynamic proxy is hosted on PC-4, while the UDDI registry and alternate loan-approver Web services are all deployed on PC-5. For UDDI registry, we chose JUDDI. JUDDI is an open source Java implementation of the Universal Description, Discovery, and Integration (UDDI) specification for Web services from the Apache Software Foundation.

As illustrated in Figure 7, client requests are made to the BPEL process on PC-2 (labeled 1), which results in the invocations to the partner Web services on PC-3 (labeled 2). Upon failure of these partner services or an invocation timeout, the adapt-ready BPEL process invokes the dynamic proxy (labeled 3). The dynamic proxy first queries the JUDDI registry on PC-5 for substitute services (labeled 4). The result of the query is used to bind the substitute service on PC-5 and forward the requests to this service (labeled 5).

4.3 Self-Healing and Self-Optimization

In order to demonstrate the autonomic behavior of the generated BPEL process and its corresponding dynamic proxy, we have programmatically altered the Loan Approver Web service deployed on PC-3 to generate faults and a delay of two seconds after a certain number of successive invocations. The successive invocations to the Loan Approver Web service are the results of requests to the BPEL process made by the client application. These requests are mapped on the X axis of the chart shown in Figure 8. As this figures shows, for the successive invocations 11 to 20, the Loan Approver Web service generates a fault for those invocations, and for the invocations 31 to 40, the Loan Approver Web service is made to delay for 2 seconds before sending back a reply to the BPEL process. The fault generation is meant to simulate a problematic Web service, a server crash, or a network outage and the delay is meant to simulate an overly loaded Web service or its corresponding host (in this case PC-3). We set the timeout duration for the Loan Approval BPEL process to 1 second and then from the client program, we made 50 successive requests to the Loan Approval BPEL process.

The two plots in Figure 8 show the request completion time for the 50 requests: one plot reflects the



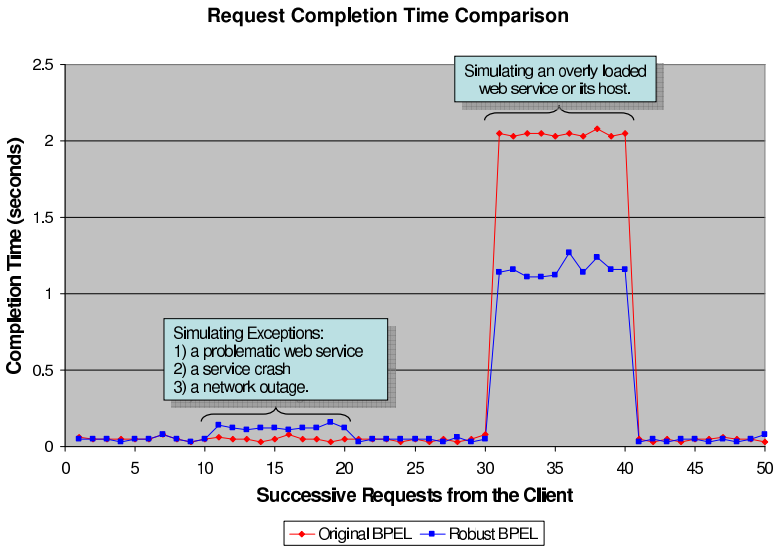


Figure 8: This chart shows the comparison between the request completion time for the original and the robust BPEL processes.

behavior of the original BPEL process and the other one reflects the behavior of the generated robust BPEL process and its corresponding dynamic proxy. According to the experiment setup, the first 10 request are completed normally as there are no fault generated or no delay is added on the execution path of these request. As expected, the average completion time for both the original and the robust sets of experiments are almost the same (about 47 milliseconds). This result indicates that in normal operation, the overhead added by the robust BPEL process is negligible.

Right after the completion of the first 10 requests, the Loan Approver Web service starts throwing exceptions for the next 10 requests. Although Figure 8 shows that the completion time for the original BPEL stays as before, but all the requests are returned with exception and the results are of no use. In other words, the original BPEL process fails its clients. The robust BPEL process, however, catches all such exceptions and uses the dynamic proxy, which resides on PC-4, to find an equivalent service. In its turn, the dynamic proxy uses the UDDI server on PC-5 and finds the substituent service also deployed on PC-5. The

plot for robust BPEL in Figure 8 shows an increase in the completion time, which is about 127 milliseconds. For many applications, the extra 80 milliseconds overhead is much more desirable than receiving a faulty response.

For the next 10 requests (21 to 30), the Loan Approver Web service goes back to its normal operation and responds to the requests without throwing exceptions. As can be seen from Figure 8, the robust BPEL uses the original Web service and in essence optimizes the completion time for these 10 requests. As illustrated in the original BPEL plot, for the next 10 requests (31 to 40), the Loan Approver Web service responds to the requests after 2 seconds of delay. As the time out in the robust BPEL process is set to 1 second, the robust BPEL process withdraws its invocations to the original Loan Approver Web services after 1 second and uses the substitute Web service. In this way, the robust BPEL process completes the request in almost half the time as that of the original BPEL process.

5 Related Work

There has been a lot of work done both in Web service monitoring [8, 23] and in adding fault tolerant to existing systems [19, 20]. In this section, we cover only those that are most related to the work presented in this paper.

Dialani et al. [11] provide an approach to enabling fault tolerance in *stateful* Web services by requiring the developer to implement an interface for rollback and checkpoint. In their approach, the service interface of the targeted Web services (reflected in the corresponding WSDL documents) must be extended to include the methods for fault tolerance. With the use of global and local fault managers, the Web services are monitored individually (by local fault managers) and in a composition (by global fault managers). Local fault managers are implemented as libraries that are dynamically bound to the service code. Checkpoints are used locally and a rollback can be initiated locally or globally after a fault is detected. The Web services have to declare their inter-dependencies so that the fault managers can control fault recovery. Further, an extension is made to the SOAP communication layer so that messages can be logged and replayed. The global fault managers interact with the services in the composition via the set of interfaces that are implemented locally. This work is complementary to ours but it focuses on *stateful* Web services while we specifically focus on *aggregate* Web services with the assumption that the partner Web services are stateless.

Birman et al. [5] propos extensions to the Web services architecture to support mission-critical applications. They propose the following five extensions; Component Health Monitoring (CHM), Consistent and Reliable Messaging(CRM), Data dissemination (DDS), Monitoring and Distributed Control (MDC) and Event notification (EVN). CHM represents new services that are used to track the health of individual Web service. Like a domain name service, CHM maps service component identifiers to health information. Changes to the state of monitored components are reported to CHM and components interested in the health of monitored components would connect to CHM. CRM involves the use of a group communication interface and an extension to TCP to achieve TCP stream replication over a set of group members. They claim that CRM offers a form of unbreakable TCP endpoint and that in conjunction with WS-RELIABILITY leads to safe handoffs without the need to persist information on a disk. CHM and CRM can be used transparently by Web services routers without the need to modify the client or server applications. MDC deals with monitoring by tracking performance metrics and other state variables and reporting them out. Unlike CHM, MDC looks at aggregated properties of the system as a whole by correlating statistic from individual components. DDS focuses on reliable streaming of data by a service to its clients. EVN is a mechanism for sending urgent one-time event notifications to the client. Similar to ours, this work aims to improve the reliability of Web serives, but it proposes extensions to the Web services architecture.

Baresi's approach [4] to monitoring involves the use of annotations that are stated as comments in the source BPEL program and then translated to generate a target monitored BPEL program. In addition to

monitoring functional requirements, timeouts and runtime errors are also monitored. Whenever any of the monitored conditions indicates misbehaviour, suitable exception handling code in the generated BPEL program handles them. This approach is much similar to ours in that monitoring code is added after the standard BPEL process has been produced. This approach achieves the desired separation of concern. This approach however requires modifying the original BPEL processes *manually*. The annotated code is scattered all over the original code. The manual modification of BPEL code is not only difficult and error prone, but also hinders maintainability. In our approach, there is no need for annotation and manual modification of the original BPEL processes.

Finally, BPELJ [6] is an extension to BPEL. The goal of BPELJ is to improve the functionality and fault tolerance of BPEL process. This is accomplishes by embedding snippets of Java code in the BPEL process. This however requires a special BPEL engine, thereby limiting its portability of BPELJ processes. The works mentioned above, although are able to provide some means of monitoring for singular or aggregate Web services, they do not dynamically replace the delinquent services once failure or extensive delay has been detected.

[**sms:** *Add a couple of sentences and mention how service selection (specifically works in ranking services) complements our work.*] We note that several work has been done in the area of service selection [22, 29], but ....

6 Conclusion and Future Work

We presented an approach to transparently adapting BPEL processes to tolerate run-time and unexpected faults and to improve the performance of overly loaded Web services. We have introduced the dynamic proxy and demonstrated how it is used to encapsulate autonomic behavior. With the use of a case study, we demonstrated the self-healing and self-optimization behavior of the dynamic proxy.

In our future work, we plan to address the following issues. First, we realize that the performance of the dynamic proxy can be improved further by using a caching mechanisim to avoid making reperitive calls to the UDDI registry. Second, substituting service implementations at runtime may lead to failures on the client-side [10]. Thus it is important to be able to detect and resolve potential integration problems from discovered equivalent services. This could include making sure that the substitute services would actually fulfill their functional requirements. Third, we realized that the task of improving fault tolerance and performance for multiple service collaborations is made even more complex if the collaborating services are *stateful*. We plan to investigate and adopt techniques such as those of [11] for this purpose. Fourth, describing what constitutes an equivalent service can be a contentious issue, there is therefore a need for a formal definition of service equivalence. Finally, we plan to study the existing ranking systems for Web services and add this logic to the dynamic proxy.

Further Information. A number of related papers, technical reports, and a download of the software developed for this paper can be found at the following URL: `http://acrl.cis.fiu.edu/`.

Acknowledgements. The authors are very thankful to Eduardo Monteiro who worked on the implementation of the generator. This work was supported in part by IBM SUR grant.

References

[1] Active Webflow. Available at URL: `http://www.active-endpoints.com/products/`.

[2] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. *Business Process Execution Language for Web Services version 1.1*. BEA Systems, International Business Machines Corporation, Microsoft Corporation, SAP

AG, and Siebel Systems., 1.1 edition, May 2003. Available at URL: ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf.

[3] B. Atkinson, T. Bellwood, M. Cahuzac, L. Clment, J. Colgrave, U. Corda, A. Czimbor, M. J. Dovey, D. Feygin, S. Garg, R. Gupta, A. Hately, B. Henry, A. Kawai, P. Macias, A. T. Manes, C. von Riegen, T. Rogers, A. Srivastava, P. Thorpe, A. Triglia, M. Voskob, and G. Zagelow. *UDDI Version 3.0.1*. OASIS, 2003. Available at URL: http://uddi.org/pubs/uddi_v3.htm.

[4] L. Baresi, C. Ghezzi, and S. Guinea. Smart monitors for composed services. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 193–202. ACM Press, 2004.

[5] K. P. Birman, R. van Renesse, and W. Vogels. Adding high availability and autonomic behavior to web services. In *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, pages 17–26, Edinburgh, United Kingdom, May 2004. IEEE Computer Society.

[6] M. Blow, Y. Goland, M. Kloppmann, F. Leymann, G. Pfau, D. Roller, and M. Rowley. *BPELJ: BPEL for Java, A Joint White Paper by BEA and IBM*. BEA and IBM, March 2004. Available at URL: http://ftpna2.bea.com/pub/downloads/ws-bpelj.pdf.

[7] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard. *Web Services Architecture*. W3C, 2004. Available at URL: http://www.w3.org/TR/ws-arch/.

[8] G. Canfora, P. Corte, A. D. Nigro, D. Desideri, M. D. Penta, R. Esposito, A. Falanga, G. Renna, R. Scognamiglio, F. Torelli, M. L. Villani, and P. Zampognaro. The c-cube framework: Developing autonomic applications through web services. In *Proceedings of DEAS'05*, Missouri, USA, May 2005.

[9] R. Chinnici, M. Gudgin, J.-J. Moreau, J. Schlimmer, and S. Weerawarana. *Web Services Description Language (WSDL) Version 2.0*. W3C, 2.0 edition, March 2004. Available at URL: http://www.w3.org/TR/wsdl20/.

[10] G. Denaro, M. Pezze, and D. Tosi. Adaptive integration of third party web services. In *in Proceeding DEAS 2005*, St. Louis, Missouri, USA, May 2005.

[11] V. Dialani, S. Miles, L. Moreau, D. D. Roure, and M. Luck. Transparent fault tolerance for web services based architectures. In *Eighth International Europar Conference (EURO-PAR'02)*, Lecture Notes in Computer Science, Padeborn, Germany, aug 2002. Springer-Verlag.

[12] E. W. Dijkstra. Structured programming. *Software Engineering Techniques, edited by Buxton and Randell (available from NATO, Brussels)*, pages 84–87, 1970.

[13] O. Ezenwoye and S. M. Sadjadi. Enabling robustness in existing BPEL processes. In *Proceedings of the 8th International Conference on Enterprise Information Systems (ICEIS-06)*, May 2006. To appear.

[14] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, and H. F. Nielsen. *SOAP Version 1.2*. W3C, 1.2 edition, 2003. Available at URL: http://www.w3.org/TR/soap12.

[15] S. Gurguis and A. Zeid. Towards autonomic web services: Achieving self-healing using web services. In *Proceedings of DEAS'05*, Missouri, USA, May 2005.

[16] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[17] D. C. Marinescu. *Internet-Based Workflow Management: Towards a Semantic Web*. Wiley-Interscience, 2002.

[18] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng. Composing adaptive software. *IEEE Computer*, pages 56–64, July 2004.

[19] L. Moser, P. Melliar-Smith, P. Narasimhan, L. Tewksbury, and V. Kalogeraki. The Eternal system: An architecture for enterprise applications. In *Proceedings of the Third International Enterprise Distributed Object Computing Conference (EDOC'99)*, July 1999.

[20] B. Natarajan, A. S. Gokhale, S. Yajnik, and D. C. Schmidt. DOORS: Towards high-performance fault tolerant CORBA. In *International Symposium on Distributed Objects and Applications*, pages 39–48, 2000.

[21] C. Peltz. Web services orchestration a review of emerging technologies, tools and standards. *Technical Paper*, January 2003.

[22] S. Ran. A model for web services discovery with QoS. *SIGecom Exch.*, 4(1):1–10, 2003.

[23] W. N. Robinson. Monitoring web service requirements. In *Proceedings of the 11th IEEE International Conference on Requirements Engineering (RE 2003)*, pages 65–74. IEEE Computer Society, September 2003.

[24] S. M. Sadjadi. *Transparent Shaping for Existing Software to Support Pervasive and Autonomic Computing*. PhD thesis, Department of Computer Science, Michigan State University, East Lansing, United States, August 2004.

[25] S. M. Sadjadi and P. K. McKinley. Using transparent shaping and web services to support self-management of composite systems. In *Proceedings of the International Conference on Autonomic Computing (ICAC'05)*, pages 88–95, Seattle, Washington, June 2005.

[26] S. M. Sadjadi, P. K. McKinley, and B. H. Cheng. Transparent shaping of existing software to support pervasive and autonomic computing. In *Proceedings of the first Workshop on the Design and Evolution of Autonomic Application Software 2005 (DEAS'05), in conjunction with ICSE 2005*, St. Louis, Missouri, May 2005. To appear.

[27] D. Sherman. Business flows with bpel4ws. *Online article*, 2005. Available at URL: `http://xml.sys-con.com/read/39780.htm`.

[28] S. Weerawarana and F. Curbera. Business process with bpel4ws: Understanding. *Online article*, 2002. Available at URL: `http://www-128.ibm.com/developerworks/webservices/library/ws-bpelcol1/`.

[29] T. Yu and K.-J. Lin. The design of qos broker algorithms for qos-capable web services. *International Journal of Web Service Research*, 1(4):33–50, 2004.