RobustBPEL2: Transparent Autonomization in Business Processes through Dynamic Proxies

Onyeka Ezenwoye and S. Masoud Sadjadi
Autonomic Computing Research Laboratory
School of Computing and Information Sciences
Florida International University
11200 SW 8th Street
Miami, FL 33199
{oezen001,sadjadi}@cs.fiu.edu

Abstract

Web services paradigm is allowing applications to interact with one another over the Internet. BPEL facilitates this interaction by providing a platform through which Web services can be integrated. However, the autonomous and distributed nature of the integrated services presents unique challenges to the reliability of composed services. The focus of our ongoing research is to transparently introduce autonomic behavior to BPEL processes in order to make them more resilient to the failure of partner services. In this work, we present an approach where BPEL processes are adapted by redirecting their interactions with partner services to a dynamic proxy. We describe the generative adaptation process and the architecture of the adaptive BPEL processes and their corresponding proxies. Finally, we use case studies to demonstrate how generated dynamic proxies are used to support self-healing and self-optimization in instrumented BPEL processes.

1 Introduction

Web services are facilitating the uptake of Service-Oriented Architecture (SOA) [3], 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 [7] can be used to define aggregate services (business processes) that constitute a number of related services (business functions). Unfortunately, these types of business processes are known to be very fragile, 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 might have been developed and hosted on heterogeneous environments, introduces new levels of complexity in management. Also, services interacting with aggregate services are often geographically scattered and communicate via the Internet. 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 [2]. It is then important to make aggregate services more resilient to the failure of their partner services.

Autonomic computing [10] 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 [9]. Autonomic attributes include self-configuration, self-optimization, self-healing, and self-protection. The focus of this work is to encapsulate *self-healing* and *self-optimizing* behavior in business processes in order to make them more resilient to failure.

We recently introduced RobustBPEL [8], a framework that provides a *systematic* approach to making existing aggregate Web services more tolerant to the failure. 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 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 extend the RobustBPEL framework to generate a proxy that *dynamically* discovers and binds to existing services. Because more appropriate services may become available after the composition and deployment of the BPEL process and its corresponding static proxy, it makes sense that upon failure or delay of any of the partner services of the BPEL process, an equivalent service can be discovered dynamically (at run-time) to serve as a substitute. 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 autonomic behavior with that of the business process [11]. This transparency is achieved by using a *dynamic* proxy that encapsulates the autonomic behavior (adaptive code).

The rest of this paper is is structured as follows. Section 2 provides a background on some related technologies and gives a brief introduction to the RobustBPEL framework. Section 3 describes the dynamic proxy and how it is generated. In section 4 we use two examples as case studies to demonstrate the feasibility and usefulness of our approach. Section 5 contains some related work. Finally, some concluding remarks and a discussion on further research directions are provided in Section 6.

2 Background

In this section, we provide some background information for Web services, BPEL, Transparent Shaping and RobustBPEL. You can safely skip this section if you are familiar with all the above technologies.

2.1 Web Services & BPEL

A Web service is a software component that can be accessed over the Internet. The goal of the Web service architecture [3] 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). The interface to the functionality provided by a Web service is described in Web services Description Language (WSDL). To make a call on these functions, a messaging protocol such as SOAP can be used.

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. Currently search engines like Google, Yahoo! and MSN are being exposed as Web services and provide functions that range from simple queries, to generation of maps and driving directions. A business process that can be derived from the aggregation of such services would be one that, for instance, generates driving directions. As illustrated by Figure 1, the process could work by integrating two service: (1) a service that retrieves the addresses of nearby businesses; and (2) a service that gets the driving directions to a given address. This business process can then be used from the on-board computer of a car to generate driving directions to the nearest gas station, hotel, etc.

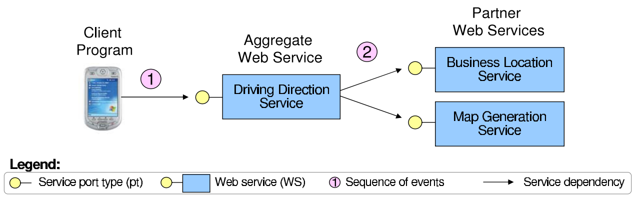
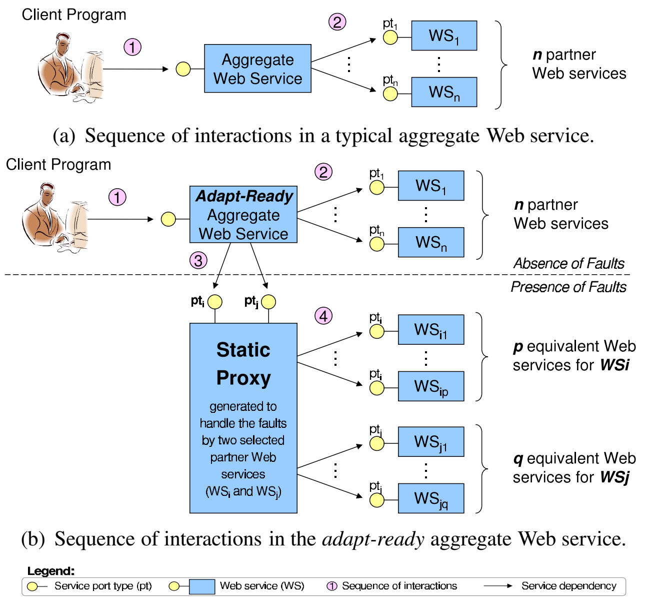


Figure 1. A Business Process that integrates remote components to create a new application that gets driving directions.

To facilitate the creation of business processes, a high-level workflow language, such as Business Process Execution Language (BPEL) [7], is often used. BPEL provides many constructs for the management of a process including loops, conditional branching, fault handling and event handling (such as timeout). To make a BPEL process fault tolerant, BPEL fault handling activities (*e.g.,* `catch` and `catchAll` constructs) can be used. We aim to separate the task of making a BPEL process more robust from the task of composing the business logic of the process.

2.2 Transparent Shaping & RobustBPEL

Transparent Shaping is a new programming model that provides dynamic adaptation in existing applications. The goal is to respond to the dynamic changes in their non-functional requirements (*e.g.,* changes request by end users) and/or environments (*e.g.,* changes in the executing environment) [11]. In transparent shaping, an application is augmented with *hooks* that intercept and redirect interaction to *adaptive code*. The adaptation is transparent because it preserves the original functional behavior and does not tangle the code that provides the new behavior (adaptive code) with the application code. By adapting *existing* applications, transparent shaping aims to achieve a separation of concerns [5]. That is, enabling the separate development of the functional requirements (the business logic) from the non-functional requirements of an application.



(a) Sequence of interactions in a typical aggregate Web service.

(b) Sequence of interactions in the *adapt-ready* aggregate Web service.

Figure 2. Architectural diagrams showing the difference between the sequence of interactions among the components in a typical aggregate Web service and its generated adapt-ready version.

RobustBPEL [8] is a framework that we developed previously as part of the transparent shaping programming model. Using RobustBPEL, we can automatically generate an *adapt-ready* version of an existing BPEL process. We note that in our previous study, we only focused on adding self-healing (fault-tolerant) behavior to existing BPEL processes.

Figure 2 shows the differences between the sequence of interactions among the components in a typical aggregate Web service and its corresponding generated adapt-ready version. In a typical aggregate Web service (Figure 2(a)), first a request is sent by the client program, then the aggregate Web service interacts with its partner Web services (*i.e.,* WS_1 to WS_n) and responds to the client. If one of the partner services fails, then the whole process is subject to failure. To avoid such situations, adapt-ready process monitors the behavior of it partners and tries to tolerate their failure.

The developer can select a set of Web service partners to be monitored. For example, in Figure 2(b) WS_i and WS_j have been selected for monitoring. The adapt-ready process monitors these two partner Web services and in the presence of faults it will forward the corresponding request to the *static proxy*. The static proxy is generated specifically for this adapt-ready process and provides the same port types as those of the monitored Web services (*i.e.,* pt_i and pt_j). The static proxy in its turn forwards the request to an *equivalent* Web service, which is “hardwired” into the code of this proxy at the time it was generated. This means that the number of choices for equivalent services are limited to those known at the time the static proxy was generated.

In this work, we make the following assumptions: (1) two services are *equivalent*, if they implement the same port type; (2) Web service partners are *stateless* and *idempotent*. A *port type* is similar to an interface in the Java programming language. It is possible for two applications to be functionally equivalent without necessarily having the exact same interface. When this occurs, a wrapper interface/service can be used to harmonize the differences in their interfaces. We show as example of this scenario in our case studies.

2.3 Why Dynamic Proxies?

Given the rapid uptake of the service oriented programming model, we expect the emergence of numerous services that are functionally equivalent and thus can be substituted. For instance, in our driving-direction example (Figure 1), if the default map generation service provided by Google fails, it should be possible to substitute this service with that of MSN, Yahoo! or Mapquest.

In this paper, we extend RobustBPEL in two directions: (1) by replacing static proxies with *dynamic* proxies that can find equivalent services at run time (described in Section 3); and (2) by adding self-optimizing behavior in existing BPEL processes, which is demonstrated using the case studies in Section 4.

3 Dynamic Proxies

In our approach, a *dynamic proxy* is a Web service that corresponds to a specific adapt-ready BPEL process and its job is to discover and bind *equivalent* Web services. In this section, we first provide an architecture that shows the high-level functionality of a dynamic proxy and its interactions with other services in the architecture. Next, we explain how the adapt-ready BPEL process is instrumented and how it interacts with the dynamic proxy. Finally, we show a high-level view of the RobustBPEL2 Generator.

3.1 High-Level Architecture

Figure 3 illustrates the architectural diagram of an application using an adapt-ready BPEL process augmented with its corresponding dynamic proxy. This figures shows the steps of interactions among the components of a typical adapt-ready BPEL process. Similar to a static proxy, the interface for the generated dynamic proxy is exactly the same as that of the monitored Web service. 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. For example, the dynamic proxy in Figure 3 has pt_i and pt_j, which are the port types of the two monitored Web services (namely, WS_i and WS_j). At runtime, if a 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 adapted BPEL process via the proxy.

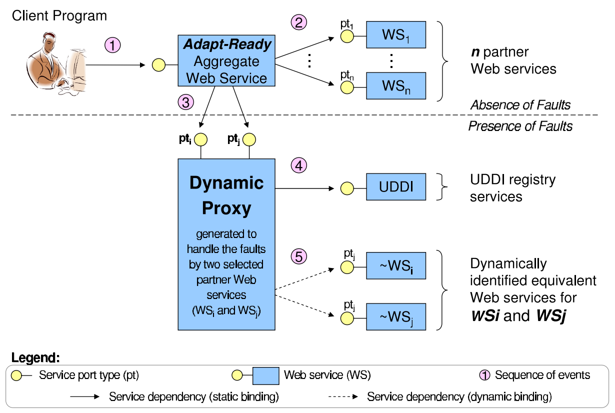

Client Program
1
2
3
4
5
Adapt-Ready Aggregate Web Service
pt_1
WS_1
pt_n
WS_n
n partner Web services
Absence of Faults
Presence of Faults
pt_i
pt_j
Dynamic Proxy
generated to handle the faults by two selected partner Web services (WS_i and WS_j)
UDDI
UDDI registry services
$\sim WS_i$
$\sim WS_j$
Dynamically identified equivalent Web services for WSi and WSj
Legend:
Service port type (pt)
Web service (WS)
Sequence of events
Service dependency (static binding)
Service dependency (dynamic binding)

Figure 3. Architectural diagram showing the sequence of interactions among the components in an adapt-ready BPEL process augmented with its corresponding dynamic proxy.

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 [9]. By replacing failed and delayed services with substitutes, the proxy service provides self-healing and self-optimization behavior to the BPEL process, thereby making the BPEL process autonomic.

3.2 Incorporating Generic Hooks inside the Adapt-Ready BPEL Processes

Following the Transparent Shaping programming model [11], we first need to incorporate some generic hooks at sensitive *joinpoints* in the original BPEL process. 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 BPEL process *sensing* and *actuating* are required and inserting appropriate code (hooks) to do so. Because a BPEL process is an aggregation of services, the most appropriate place to insert interception hooks is at the interaction joinpoints (*i.e.,* the `invoke` instructions) [11]. The monitoring code we insert is in the form of standard BPEL constructs to ensure the portability of the modified process.

We adapt the BPEL process by identifying points in the process at which external Web services are invoked 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 from the Web service infrastructure. The following snippet BPEL code (Figure 4) 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.

```
1. <invoke name="InvokeWSi"
2.         partnerLink="..."
3.         portType="pti"
4.         operation="operation1"
5.         inputVariable="..."
6.         outputVariable="...">
7. </invoke>
```

Figure 4. An unmonitored invocation.

The invocation showed in Figure 4 is identified and wrapped with monitoring code. The code in Figure 5 shows what 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 5-14 and 15-19 respectively in Figure 5). A `catchAll` fault handler is added (lines 6-13) to the `faultHandlers` to handle any faults generated as a result of the invocation of the partner Web service. The fault-handling activity is defined in lines 7-12, which basically forwards the request to the dynamic proxy. 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.

For the event handler, an `onAlarm` event handler (lines 16-18) is used to 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. A `throw` activity is inserted inside the `onAlarm` event handler (line 17) as the action that is carried out upon the timeout. If the partner service fails to reply within the time stipulated in the timeout event, the `throw` activity generates a standard BPEL `forcedTermination` fault. This fault

```
1. <scope>
2.   <!-- linking instructions -->
5.   <faultHandlers>
6.     <catchAll>
7.       <invoke name="InvokeProxy"
8.               partnerLink="..."
9.               portType="pti"
10.              operation="operation1"
11.              inputVariable="..."
12.              outputVariable="..."/>
13.    </catchAll>
14.  </faultHandlers>
15.  <eventHandlers>
16.    <onAlarm for="'PT1S'">
17.      <throw faultName="forcedTermination"/>
18.    </onAlarm>
19.  </eventHandlers>
20.  <invoke name="InvokeWSi"
21.          partnerLink="..."
22.          portType="pti"
23.          operation="operation1"
24.          inputVariable="..."
25.          outputVariable="..."/>
26.</scope>
```

Figure 5. A monitored invocation.

forces the monitored invocation to terminate. The generated `forcedTermination` fault is then caught by the fault handler and the proxy service is invoked (lines 7-12) as a substitute.

3.3 Interaction of Dynamic Proxy with the Registry Service

When the dynamic proxy is invoked upon failure of a monitored service, the proxy makes queries against the registry service to find equivalent services. At runtime, any service provider can publish new equivalent services with the registry, which can potentially substitute a failed service in the future.

The registry technology used in the RobustBPEL2 framework is the Universal Description, Discovery and Integration protocol (UDDI), 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. The registry allows for three categories of information to be published: (1) *white pages* that contain contact information such as the name, address and 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).

In order to adequately categorize services in a UDDI registry, certain conventions have to be adhered to. The method of classification we use focuses on registering services based on the information in their WSDL descriptions, in other words, mapping WSDL to UDDI. Information about the WSDL `service` and `port` are stored under components of the UDDI data model. Data registered from the WSDL includes the URL for each service port. The dynamic proxy makes queries to the UDDI registry via the API provided by JUDDI, which is an open source Java implementation of the UDDI specification. The query term is fixed since with the port types of the monitored services is known during adaptation. At this stage of our work, no selection criteria is used when multiple services are discovered, although some selection policies can be easily incorporated into the proxy to introduce some added quality-of-service.

3.4 The Generation Process in RobustBPEL2

As part of RobustBPEL2, we developed the `RobustBPEL2 Generator` that automatically generates the adapt-ready version of a given BPEL process and its associated dynamic proxy. The input to this generator is a configuration file. Figure 6 shows the contents of a configuration file that has all the required information: lines 2-10 specify the input needed for the generation of the adapt-ready BPEL process, while lines 11-36 specify that for the generation of the proxy. As illustrated in Figure 7, first, the `Parser` separates the information needed for generating adapt-ready BPEL process and for generating the dynamic proxy and sends them to the corresponding compilers. Next, each of these two generators uses the information provided by the parser and retrieves the required files from the local disk and starts its compilation process.

The `Adapt-Ready BPEL Compiler` retrieves the original BPEL process using the location of the original BPEL file, which is stated in line 5. It then uses the `<adapt>` element (lines 7-9) to find out the names of the invocations to be monitored. An `<invoke>` element (line 8) with a "*" as the value of the name attribute declares that all invocations should be monitored. With all this information, the `Adapt-Ready BPEL Generator` is ready to starts its compilation process.

The `Dynamic Proxy Compiler` gets the location of the proxy template is on line 12 and the necessary information about every monitored service type (classified by portType) from the `<substitutes>` tags (lines 19-34). The information needed to generate the proxy code to find and bind to services that implement the portType pt_i is listed within the `<service>` elements of lines 20-30 and the operations of every monitored invocation of the portType are listed within the `operations` element. Next, the The `Dynamic Proxy Generator` finds out about the location of the binding stubs for services of the portType are in line 29 (this stub package is associated with the proxy as a Java

```
1. <generator>
2.   <bpel>
3.     <processName name="processName"/>
4.     <targetNamespace name="http://..."/>
5.     <inputFile name="originalBPEL.bpel"/>
6.     <outputFile name="adaptReadyBPEL.bpel"/>
7.     <adapt>
8.      <invoke name="*" timeout="'PT1S'"/>
9.     </adapt>
10.  </bpel>
11.  <proxy type="dynamic">
12.    <templateFile name="DynamicProxy-Template.java"/>
13.    <outputFile name="DynamicProxy.java"/>
14.    <wsdlFiles>
15.      <wsdl name="wsins" url="WSi.wsdl"/>
16.      <wsdl name="wsjns" url="WSj.wsdl"/>
18.    </wsdlFiles>
19.    <substitutes>
20.      <service name="WSiService"
21.               portType="pti"
22.               wsdl="lns:wsins">
23.        <operations>
24.          <operation name="operation1"
25.                     port="Port"/>
26.        </operations>
27.        <query businessKey="BK1"
28.               serviceName="WSiService"/>
29.        <stub name="stub.wsi"/>
30.      </service>
31.      <service name="WSjService"
32.        ...
33.      </service>
34.    </substitutes>
35.    <package name="processName.proxy.dynamic"/>
36.  </proxy>
37.</generator>
```

Figure 6. Configuration file for the generator

import statement). It then gets the information needed to query the registry for services that implement the portType from the `<query>` tag (lines 27-28). Finally, it gets the the package for the proxy class from Line 35 (the broker binding stubs are part of this package) and compiles the dynamic proxy.

4 Case Studies

In this section, we use two case studies to demonstrate the self-healing and self-optimization behavior of the generated BPEL processes and their respective dynamic proxies. For each case study, we start by describing the application, then we present the configuration of the experiment environment. Finally, we show the results of the experiment.

4.1 The Google-Amazon Process

The Google-Amazon business process integrates the Google Web service for spelling suggestions with the Amazon E-Commerce Web service for querying its store catalog. The business process takes as input a phrase (keywords) which is sent to the Google spell-checker for corrections. If any word in the input phrase is misspelled, the Google spell-checker sends back as reply the phrase with the misspelled words corrected (the phrase is unchanged if the spellings are correct). The reply from the Google service is used to create keyword search of the Amazon bookstore via the Amazon Web service.

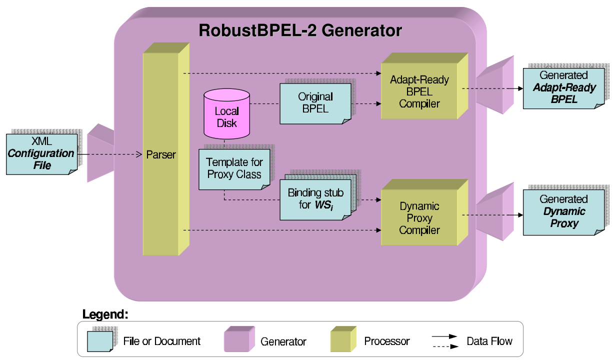


Figure 7. Inputs and outputs of the dynamic proxy generator.

From this original Google-Amazon process, we used the generator to generate the adapt-ready process. For this adaptation we have selected to have the generator only adapt the invocation of the Google spell-checker. We then found another publicly available Spell-checker Web service from Cydne to act a substitute for the Google service. There is a slight difference between the interfaces of the Google and Cydne spell-checkers. We used a wrapper Web service for the Cydne service in order to harmonize the interfaces.

4.1.1 The Experiment

As illustrated in Figure 8, client requests are made to the BPEL process (labeled 1), which results in the invocations to the Google Web services (labeled 2). To simulate the unavailability of the Google service, we changed the URL of the service from within the Google-Amazon process, to point to a non-existent address. Thus upon the imminent failure of the invocation for the Google service, the adapt-ready BPEL process invokes the dynamic proxy (labeled 3). The dynamic proxy first queries the JUDDI registry for substitute services (labeled 4). As a result of the query, it finds the wrapper Web service for the Cydne spell-checker. The proxy then binds to the wrapper service, which in turn binds to the Cydne spell-checker with the input keywords (labeled 5 and 6, respectively). The result of this invocation is sent back to the adapt-ready Google-Amazon process and then used as input to query the Amazon store service. For example, we used "Computer Algorthms" as input keyword to the process, Google (or the wrapper) corrected it to "Computer Algorithms", and Amazon found this book: "Bruce Schneier, Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition".

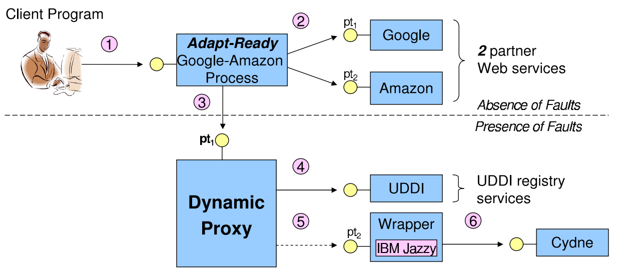


Figure 8. The sequence of interactions among the components in the Google-Amazon case study.

4.2 The Loan Approval Process

The Loan-Approval process is a commonly used sample BPEL process. The Loan-Approval BPEL process is an aggregate Web service composed of two other Web services: a low-risk assessor service (LoanAssessor) and a high-risk assessor service (LoanApprover). The Loan-Approval process implements a business process that uses its two partner services to decide whether a given individual qualifies for a given loan amount. Both the business process and the risk assessment services are deployed locally.

4.2.1 The Experiment

As illustrated in Figure 9, client requests are made to the BPEL process (labeled 1), which results in the invocations to the partner Web services (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 for substitute services (labeled 4). The result of the query is used to bind the substitute service and forward the requests to this service (labeled 5).

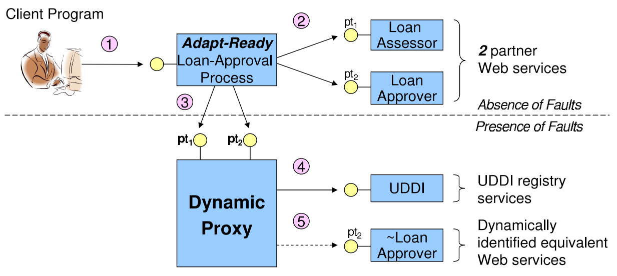


Figure 9. The sequence of interactions among the components in the Loan Approval case study.

4.2.2 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 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 10. As the plot for the original BPEL 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. We set the timeout duration for the Loan Approval BPEL process to 1 second.

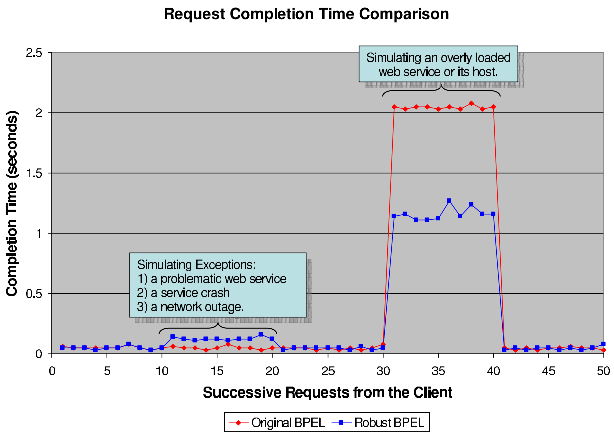


Figure 10. This chart shows the comparison between the request completion time for the original and the robust BPEL processes.

Figure 10 plots the request completion time for the two sets of experiments. According to the experiment setup, the first 10 request are completed normally and 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 10 shows that the completion time for the original BPEL stays as before, all the requests are returned with exception. The robust BPEL process, however, catches all such exceptions. The plot for robust BPEL in Figure 10 shows an increase in the completion time, which is about 127 milliseconds.

For the 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 Web services 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

Baresi's approach [1] 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. This approach requires modifying the original BPEL processes *manually* and 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.

Charfi et al [4] use an aspect-based container to provide middleware support for BPEL. The process container is the runtime environment for the BPEL process. All interactions go through the container which plugs in support for non-functional requirements. Aspects specify what and how SOAP messages can be modified to add, for instance, security information to the header. This framework is different form our because it requires a purpose built BPEL engine. Also, the adaptation is done at a much lower level (the messaging layer).

Erradi et al. [6] provide reliability through a policy driven middleware called Web Services Message Bus (wsBus), which is used to transparently enact recovery actions. The wsBus intercepts the execution of composite services and transparently provides recovery services based on an extensible set of recovery policies. The wsBus also enforces SLA agreements. This approach is modular and separates the business logic of the process from the QoS requirements, however, adaptation is done at a much lower messaging layer.

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.

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 case studies, 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, substituting service implementations at runtime may lead to failures on the client-side. Thus it is important to be able to detect and resolve potential integration problems from discovered equivalent services. Also, discovered services may fail so we aim to augment the proxy with a mechanism to deal with such problems. Finally, rather than the current specific proxies, a *generic* proxy that has a standard interface and works for all monitored services would make life much simpler for developers.

References

[1] 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.

[2] 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.

[3] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard. *Web Services Architecture*. W3C, 2004.

[4] A. Charfi and M. Mezini. An aspect based process container for BPEL. In *Proceedings of The First Workshop on Aspect-Oriented Middleware Developement*, Genoble, France, November 2005.

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

[6] A. Erradi and P. Maheshwari. wsBus: QoS-aware middleware for relaible web services interaction. In *IEEE International Conference on e-Technology, e-Commerce and e-Service*, Hong Kong, China, 2005.

[7] O. Ezenwoye and S. M. Sadjadi. Composing aggregate web services in BPEL. In *Proceedings of The 44th ACM Southeast Conference*, Melbourne, Florida, March 2006.

[8] 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.

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

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

[11] S. Masoud 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)*, Seattle, Washington, June 2005.