# A Proxy-Based Approach to Enhancing the Autonomic Behavior in Composite Services

Onyeka Ezenwoye and S. Masoud Sadjadi
Autonomic Computing Research Laboratory
School of Computing and Information Sciences, Florida International University
11200 SW 8th St., Miami, FL 33199
Tel: 305-348-1835 Fax: 305-348-2336
Email: {oezen001,sadjadi}@cs.fiu.edu

***Abstract*— Web services paradigm is allowing applications to electronically interact with one another over the Internet. The business process execution language (BPEL) takes this interaction to a higher level of abstraction by enabling the development of aggregate Web services. However, the autonomous and distributed nature of the partner services in an aggregate Web service present unique challenges to the reliability of the composite services. In this paper, we present an approach where *existing* BPEL processes are automatically instrumented, so that when one or more of their partner services do not provide satisfactory service (*e.g.,* because of a service being overwhelmed, crashed, or because of a network outage), the request for service is redirected to a *proxy Web service*, where the failed or slow services are replaced by substitute services.**

***Index Terms:* Web service monitoring, BPEL, separation of concerns, static proxies, dynamic proxies, autonomic computing, self-healing, self-optimization, dynamic service discovery.**

## I. Introduction

Web services are facilitating the uptake of service-oriented architecture (SOA) [1], allowing organizations to electronically interact with each 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. Figure 1 depicts an example aggregate service, the Sales service, that involves four functional units. In this example, the activity of processing a purchase order from a customer involves the accounting, production, inventory and delivery services. High-level workflow languages such as business process execution language (BPEL) [2], [3] can be used to define aggregate services (business processes) that constitute a number of related services [4]. Unfortunately, these types of business processes are known to be very fragile. According to [5], about 80 percent of the total amount of time used in developing business processes is spent in exception management.

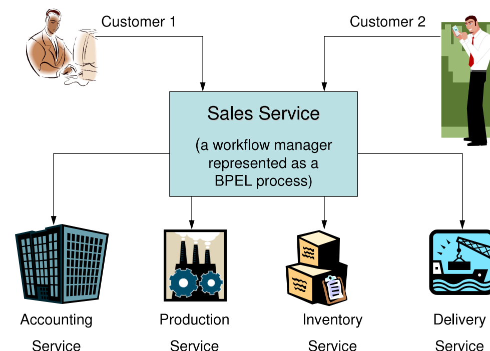



Fig. 1. An example Sales aggregate service.

The integration of multiple services, which potentially can be developed and hosted on heterogeneous environments by third-party service providers, introduces new levels of complexity in the management of composite services. Moreover, 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 and high-performance nature of some composite services require them to continue working satisfactorily in the face of failure or low performance of their constituent parts [6], [7]. It is then important to make aggregate services more resilient to the failure of their partner services.

*Autonomic computing* [8] 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 [8]. 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 their partner services (self-healing behavior), as well as to optimize their performance based on, for example, the response time of their partner services (self-

S. Masoud Sadjadi is the corresponding author for this paper. He can be reached at sadjadi@cs.fiu.edu.

This work was supported in part by IBM and the National Science Foundation (grants OISE-0730065, OCI-0636031, IIS-0552555, and HRD-0317692)

optimization behavior).

In this paper, we present a framework, which we call TRAP/BPEL, that enables us to systematically and automatically instrument BPEL processes so that when one or more of their partner services do not provide satisfactory service (*e.g.,* because of a service being overwhelmed, crashed, or because of a network outage), the request for service is redirected to a *proxy Web service*, where the failed or slow services are substituted by some available and more responsive "equivalent" services. To achieve this, events such as faults and timeouts are monitored from within the instrumented BPEL processes. When a fault or a timeout occurs, the request for service is forwarded to the associated proxy, which in its turns sends the request to an available equivalent service.

Without compromising the general application of our approach and for the sake of simplicity, in this paper we assume that *equivalent* services are those that implement the same interface (and consequently implement the same business logic) as that of the monitored services. So, when two Web services implement the same port type, we assume that they provide the same functionality and they may be different only in the quality of the service they provide. We realize that this assumption may seem too simplistic and unreal considering the proliferation of semantically different Web services that are exposed with the same interface and vice versa. However, relying on the substantial ongoing research in this area, we expect that in the near future, standard organizations will specify standard service interface definitions for different business domains that different service providers can use to expose their semantically equivalent services through the corresponding specified standard interfaces [10]. In the case where the interfaces of semantically equivalent services are not the same, all needs to be done is to go through an extra step of mapping the interfaces of the equivalent services to those of the monitored services. For example, using the adapter design pattern [11], one can expose the equivalent services with the same interface to those of the monitored services.

TRAP/BPEL offers three proxy services: *static* [12], where substitute services are known and fixed at development time, *dynamic* [13], where substitute services are discovered during runtime, or *generic* [14], where there is no difference between the original partner services and their substitutes. Each of these proxies have their own applications. The *static* proxies are bound to a limited number of alternative Web services; however, the static binding avoids the unnecessary runtime overhead, making it attractive for applications that do not need the flexibility of the dynamic proxies. The *dynamic* proxies, on the other hand, enable dynamic discovery and binding of substitute partner Web services. Dynamic proxies while may add more runtime overhead, but they provide more flexibility for applications that would rather to delay discovery of substitute services until runtime in the hope of taking advantage of a more updated (and potentially better) pool of alternative services. Finally, the *generic* proxies intercept all the invocations to the monitored partner Web services and may choose to forward the invocations to substitute services even before the original service fails. Unlike static and dynamic proxies, generic proxies can optimize the performance of composite services without wasting any time on invoking slow and unavailable partner Web services.

The major contributions of the work presented in this paper are summarized as follows. First, our approach to incorporating the autonomic behavior in BPEL processes does not require the processes to be redeveloped from scratch nor there is a need for any manual modification to the code of the *existing* processes. Second, the adaptation is done in a manner that the code for business logic is kept separated from the code for robustness of the process to avoid these different concern to result in entangled code that is difficult to maintain and evolve. Third, depending on the non-functional requirements[1] of the application, the proxy service incorporating the autonomic behavior may be chosen to be static, dynamic, or generic. In this paper, we describe the *generative* adaptation process, the architecture of the automatically instrumented BPEL processes, and their corresponding proxy services. We use a number of case studies to demonstrate how the automatically instrumented BPEL processes and their corresponding static and dynamic proxy services interact to support self-healing and self-optimization in existing BPEL processes.

The rest of this paper is is structured as follows. In Section II, we provide a background on some related technologies. In Section III, we introduce reliability in composite services and survey possible approaches to providing reliability in composite services. We finish this section by identifying where TRAP/BPEL fits in the introduced taxonomy. In Section IV, we provide an overview of our approach to instrumenting BPEL processes. In Sections V, VI, and VII, we introduce the static, dynamic, and generic proxies, respectively. In section VIII, we use three case studies to demonstrate the feasibility and usefulness of our approach. Section IX contains some related work. Finally, some concluding remarks and a discussion on further research directions are provided in Section X.

## II. Background

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

### A. Web Services

A Web service is a software component that can be accessed over the Internet. The goal of the Web service

[1] *Functional requirements* describe the interaction between the system and its actors (e.g., end users and other external systems) independent of its implementation while *non-functional requirements* are those aspects of the system that are not directly related to the functional requirements (e.g., QoS, security, scalability, performance, and fault-tolerance).

architecture [1] 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). To facilitate flexibility and interoperability, Web services are described using a standard, machine-readable, XML-based language called *Web Service Description Language* (WSDL) [15]. This service description provides the details necessary to interact with the service, including message formats that detail the operations, transport protocols, and location [16]. Finally, interaction with Web services is achieved through SOAP [17] messaging.

### B. BPEL

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). 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 2, 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.

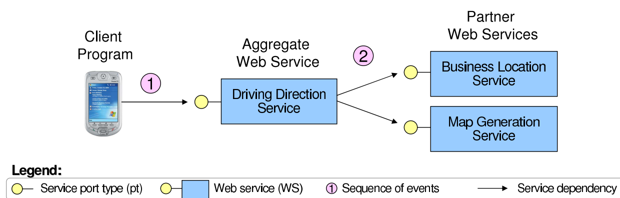



Fig. 2. A Business Process that integrates remote components to create a new application that gets driving directions.

To facilitate the creation of business processes, a high-level workflow language, such as Business Process Execution Language (BPEL) [18], 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.

### C. Web Services Protocol Stack

As illustrated in Figure 3, the key to achieving interoperability in Web services is a layered architecture. The dependency relation between each two adjacent layers is top-down. While the process layer (BPEL) takes care of the composition of Web services, the service layer (WSDL) provides a standard for describing the service interfaces. At the messaging layer (typically SOAP), the operations defined in the service layer are realized as two related output and input messages, which serialize the operation and its parameters. At the bottom of the stack is the transport layer (typically HTTP) that facilitates the physical interaction between the Web Services. Although Web services are independent of transport protocols, HTTP is the most commonly used protocol for Web service interaction. Except for the transport layer, all the protocols in the other layers are typically based on XML.

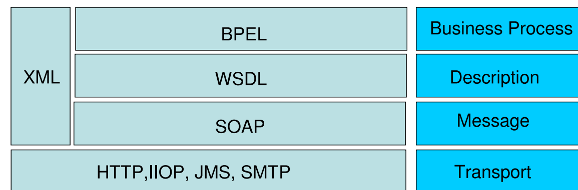



Fig. 3. The Web services stack.

### D. Transparent Shaping

*Transparent Shaping* is a new programming model that provides dynamic adaptation in existing applications [19]. 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). 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 [20]. That is, enabling the separate development of the functional requirements (the business logic) from the non-functional requirements of an application.

## III. Reliability in Composite Services

The goal of *fault-tolerance* is to improve *dependability* in a *system* by enabling it to perform its intended functions in the presence of a given number of faults [21]. There exists several definitions of dependability [22]. These definitions often depend on the attributes (*e.g.*, availability, reliability, and safety) of the system that are being defined as a criterion to decide whether or not a system is dependable at a given time. The attributes defined may depend on the intended use of the system [22].

In general, dependability is based on the notion of *reliance* in the context of interacting components. It associates to the relation *depends upon*, where a component

*A* depends upon a component *B*, if the correctness of *B's* service delivery is necessary for the correctness of *A's* service delivery [22]. This relationship is typical of composite services since they are entirely dependent on their interactions with their partner services. An error may propagate from a partner to the composite services; thereby, creating new errors. As illustrated in Figure 4, an error can propagate from component *A* to component *B*, since *B* receives service from *A*. The error propagation happens when an error reaches the service interface of component *A* and service delivered by *A* to *B* becomes incorrect. The failure of *A* becomes an external fault to *B* and propagates the error into *B* through its use interface [22].

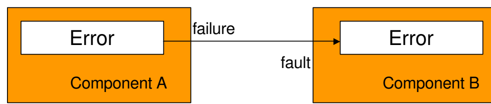



Fig. 4. Error Propagation [23]

Our work focuses on the *reliability* attribute of dependability with a specialization on *robustness* as a *secondary attribute*. Avizienis [22] defines reliability as the continuity of *correct* service and defines robustness as dependability with respect to *external* faults. Reliability is a key requirement for building dependable systems. The means to achieving dependability can be classified into four groups: (1) *fault avoidance*, through rigorous design and implementation to prevent the occurrence or the introduction of faults; (2) *fault removal*, through verification, validation and diagnosis to reduce the number or the severity of faults; (3) *fault forecasting*, to estimate the presence and consequences of faults; and (4) *fault tolerance*, to provide correct service in spite the presence of faults [24], [25]. These techniques that ere applied at development time are not sufficient enough for ensuring the reliability of composite Web services that are expected to dynamically discover and assemble components, configure themselves, and operate securely and reliably in a completely automated manner. This calls for the development of new reliability techniques that introduce *autonomic* functionality to address these challenges [6], [25].

### A. Reliability Techniques

New reliability techniques for service compositions can be developed at four layers. Figure 5 shows the different layers at which reliability techniques can be applied.

**Service provider layer.** At this level, reliability focuses on the service hosting environment. Here, reliability can be achieved by techniques that provide redundancy of computation and data, load sharing to improve performance and fault tolerance, and clustering which interconnects multiple servers to avoid single point of failure [25].

**Transport layer.** At this level, the focus is on implementing reliable messaging for Web services at the

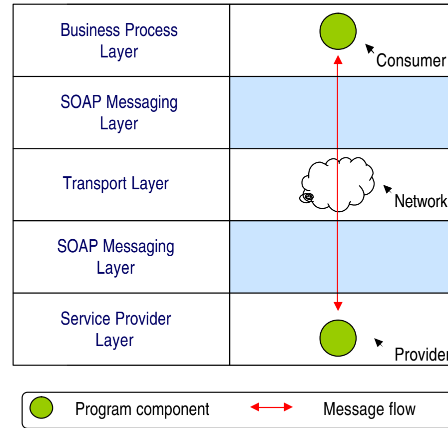



Fig. 5. Layers to apply reliability techniques.

transport layer. SOAP messaging can take different forms of reliability depending on the underlying transport service [26]. Therefore, techniques in this layer center on using message-oriented middleware (MOM) [27] to ensure reliability and robustness of message traffic. Example technologies are reliable transport protocols such as HTTPR [28] and messaging infrastructures such as IBM WebsphereMQ [29], which have built-in transactional support for business processes [25].

**SOAP messaging layer.** Addressing reliability at this layer focuses on extending SOAP messages to include reliability properties that allow messages to be delivered reliably between services in the presence of component, system, or network failures. Using recently proposed SOAP-based protocols such as WS-Reliability [30], extended SOAP messages can carry relevant reliability information. However, the extended SOAP messages must be understood and supported by a messaging infrastructure [25], [26]

**Business process layer.** Reliability at this layer aims to provide dependable composition of Web services through advanced failure handling and compensation-based transaction protocols [25], [31]. Efforts in this layer can be categorized into two groups; *language-based* and *non language-based* approaches. *Language-based* techniques provide advanced failure handling and adaptability by augmenting the process logic with additional language constructs, while *non-language* based approaches focus specifically on the process supporting infrastructure such as the execution engine.

Our work fits into this category by enabling adaptability in BPEL process to address the concerns raised above. One might argue that BPEL should be extended with constructs to handle those concerns. However, this would increase the complexity of the language and it is also against the principle of separation of concerns [32]. Constructs for specifying exceptional behavior and recovery actions should be modularized and externalized and not scattered and tangled with the service implementation. Entangling the logic for exceptional behavior and recovery actions

with the business logic of the application negatively impacts maintainability and adaptability. To address these requirements for adaptable BPEL process execution, we propose an approach that uses the transparent shaping programming model to transparently adapt their behavior. By *transparent* we mean that the adaptation preserves the original behavior of the process and does not tangle the code for *autonomic* behavior with that of the business process and its original functionality. Adaptive code is encapsulated in external components that use a set of extensible recovery policies (e.g., retry, skip, and use equivalent service) to declaratively specify how to handle exceptional behavior and how to recover from typical failures.

### B. Failure handling techniques

There are several factors that can cause failure in the execution of the workflows that define composite services in distributed environments, they include network failure, resource overload, or non-availability of required components. It is therefore important for workflow management systems to be able to identify and handle failures and support reliable execution in the presence of concurrency and failures. Workflow failure handling techniques are classified into two groups, *task-level* and *workflow-level* [33], [34]. Figure 6 shows the different techniques in each of these two groups.

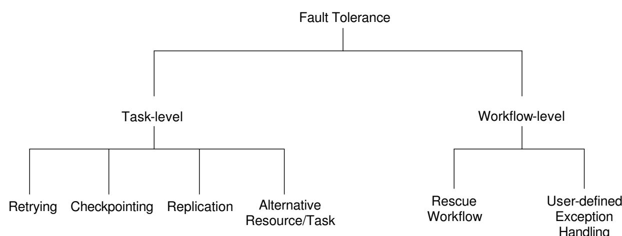



Fig. 6. A workflow fault tolerance classification.

**Task-level.** Task-level techniques mask the effects of the execution failure of individual tasks in the workflow. Task-level techniques include *retry, alternate resource, alternate task, checkpoint/restart* and *replication*. The *retry* technique tries to execute the same task on the same resource after failure. The *alternate resource* technique (also known as migration) submits failed task to another resource while the *alternate task* technique executes another implementation of a certain task if the previous one failed. The *checkpointing* technique attempts to continue workflow execution from the point of failure, this may involve moving failed tasks transparently to other resources. The *replication* technique runs the same task simultaneously on different resources to ensure successful task execution provided that at least one of the replicas does not fail [33], [34].

**Workflow-level.** Workflow-level techniques manipulate the workflow structure such as the execution flow to deal with erroneous conditions. Workflow-level techniques include *user-defined exception handling* and *rescue workflow*. *User-defined exception handling* allows users to specify a fault-handling behavior for a certain failure of a task in the workflow; the fault-handling strategy may be specified as part of the process logic. The *rescue workflow* technique executes a rescue workflow, which performs a set of recovery actions/tasks. The rescue workflow can be used, for instance, where the failure was due to the lack of disk space that can be reclaimed or in cases where totally new resources need to be assigned for continued execution [33], [34].

Our current framework uses task-level techniques to support reliability in existing BPEL processes.

## IV. INSTRUMENTING BPEL PROCESSES

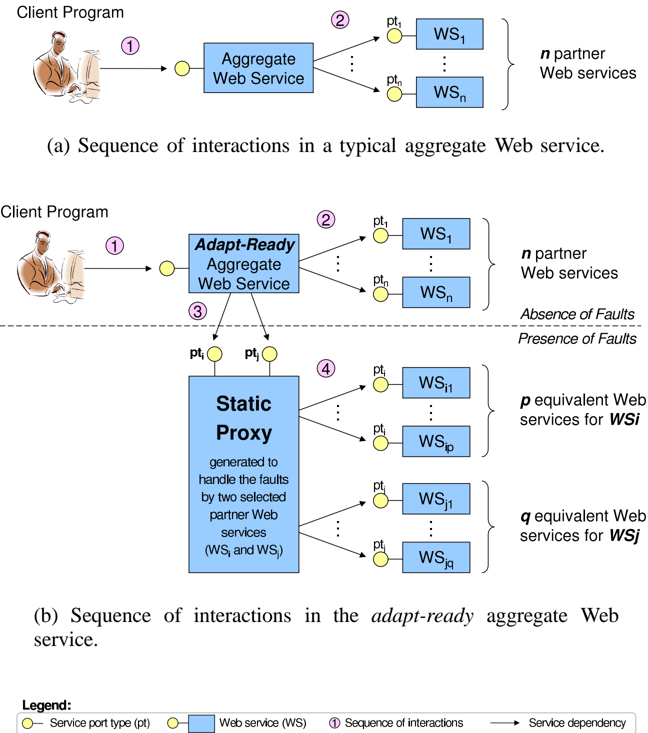
Our approach to making an existing BPEL process robust involves monitoring the invocation of partner services from within the BPEL process. Events such as timeouts and faults are monitored and upon the occurrence of such events, a proxy Web service is invoked to find and replace the faulty services. By automatically modifying existing BPEL processes, using a generator software that we developed, our approach aims to achieve a separation of concerns [20]. That is, enabling the separate development of the process's functional requirements (the business logic) from the non-functional requirements (self-healing and self-optimization, in this work). This allows the initial developer of the BPEL process, which could be a business analyst, to focus only on the functional requirements of the process and compose it as such and leave the non-functional concerns to our generative framework.

Following the Transparent Shaping programming model [19], 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) [35]. The monitoring code we insert is in the form of standard BPEL constructs to ensure the portability of the modified process. As part of the TRAP/BPEL framework, we developed a *Generator* that automatically generates the instrumented version of a given BPEL process.

## V. STATIC PROXIES

The job of a proxy Web service is to discover and bind *equivalent* Web services that can substitute for the monitored services. Equivalent services can be discovered either at design time (static discovery). During static discovery, services that can substitute for the monitored service are noted and tightly associated with the code for the proxy.

Figure 7 provides architectural diagrams showing 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 7(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.




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

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

Fig. 7. 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.

As monitoring all the partner Web services might not be necessary, the developer can select only a subset of Web service partners to be monitored. For example, in Figure 7(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. 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.

In this paper, 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. 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.

## VI. Dynamic Proxies

Figure 8 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. The main difference between the dynamic and static proxies is that the dynamic proxy is augmented a mechanism for discovering substitute services at runtime, rather that being statically bound to a limited set of substitute services.

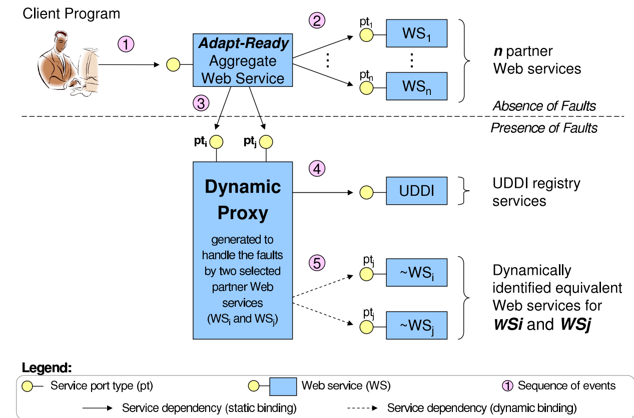



Fig. 8. Architectural diagram showing the sequence of interactions among the components in an adapt-ready BPEL process augmented with its corresponding *dynamic* proxy.

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 TRAP/BPEL 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).

There are four key components in the UDDI data model: businessEntity, businessService, bindingTemplate and tModel. The information published in the white, yellow and green pages are captured under businessEntity, businessService and bindingTemplate, respectively. The tModel (or technical model) is used to represent service taxonomies as well as store metadata about a service, such as the location of its WSDL [36].

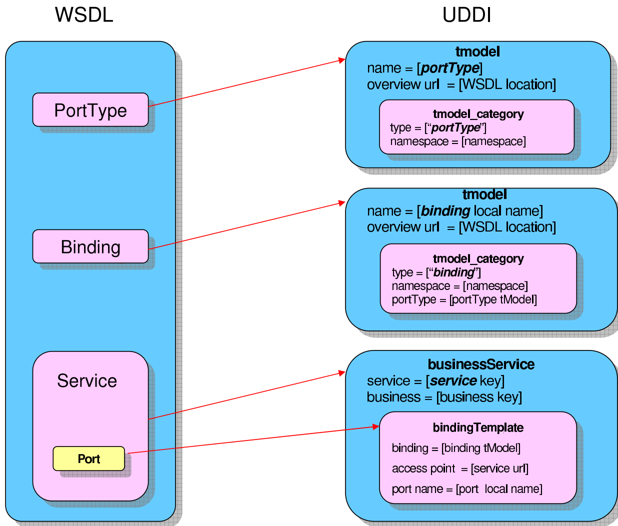



Fig. 9. Mapping WSDL to UDDI.

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.

## VII. Generic Proxies

A *generic* proxy can interact with one or more adapt-ready BPEL processes [14]. Some behavioral policy is used in the proxy to guide the adaptive behavior for each monitored service. Figure 10 illustrates several adapt-ready BPEL processes that are assigned to one generic proxy, which augments the BPEL processes with self-management behavior. The generic proxy uses a look-up mechanism to query a registry service at runtime to find out about available services. But unlike the static and dynamic proxies , the generic proxy has a standard interface which bears no relation to the interfaces of the monitored services. The generic proxy has as interface $pt_g$ that is able to receive requests for any monitored Web service (*e.g.*, $WS_{11}$ and $WS_{kn}$).

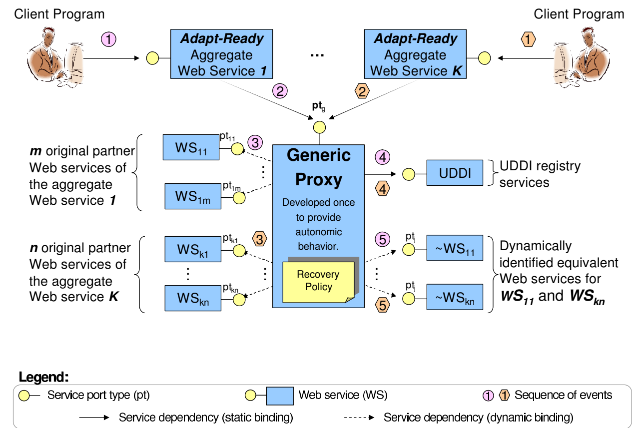



Fig. 10. Architectural diagram showing the sequence of interactions among the components in TRAP/BPEL

The generic proxy can provide self-management behavior either common to all adapt-ready BPEL processes or specific to each monitored invocation using some high-level policies. These high-level policies are specified in a configuration file that is loaded at startup time into the generic proxy. We plan to allow runtime modification to these high-level policies in future versions of our TRAP/BPEL framework. Figure 11 shows an example policy file where each unique monitored invocation can have a policy specified under a `service` element. The `InvokeName` element (line 4) has a value that uniquely identifies a monitored invocation in an adapt-ready BPEL process. The generic proxy checks all intercepted invocations and tries to match these invocations with the specified policies. If it finds a policy for that invocation, the proxy behaves accordingly, otherwise it follows its default behavior.

If a policy exists, the generic proxy may take one of the following actions according to the policy: (1) invoke the service being recommended in the policy (line 6); (2) find and invoke another service to substitute for the monitored service; and (3) retry the invocation of the monitored service in the event of its failure (line 10). The policy also specifies the time interval between retries (line 12). The

```
1. <Policy>
2.  <Service>
3.   <!--a unique name for monitored invocation-->
4.   <InvokeName value="WS-Invoke"/>
5.   <!--WSDL for a default Web service substitute-->
6.   <WsdlUrl preferred="true"value="http://./WS.wsdl"/>
7.   <!--timeout value for the monitored invocation-->
8.   <Timeout seconds="2"/>
9.   <!--the number of times to retry failed invoc.-->
10   <MaxRetry value="2"/>
11   <!--time to wait between retries-->
12   <RetryInterval seconds="5"/>
13  </Service>
14. <Service>
15.  ...
16. </Service>
17.</Policy>
```

Fig. 11. A portion of a policy file for the generic proxy.

default behavior of the proxy is to consult the registry to find and invoke an appropriate a service that implements the same inteface as the monitored invocation.

Figure 12 provides a section of the WSDL of the generic proxy. As can be seen from its description, the interface for the proxy has two operations: `genericInvoc` and `extract` (lines 22-25 and 26-29 respectively). The former is used to receive an intercepted request and the latter is used to provide the information about the reply, respectively. The input message to the `genericInvoc` operation (lines 1-6) has four parts: (1) `invokename`, which is used to identify the monitored service; (2) `porttype`, which identifies the port type of the monitored invocation (this variable is the unique key used to query the UDDI registry for services that implement the same interface); (3) `operation`, which identifies the exact operation of the port type being called; and (4) `variables`, which contains the serialized input message for the monitored service. When the proxy `genericInvocation` operation is called with the `genericInputMessage`, the proxy can identify which service is being intercepted and can discover the necessary details about the invocation. The proxy can then take one of the several actions, specified in the policy file, to monitor and adapt the intercepted call.

For dynamic service invocation, the generic proxy uses the Web Service Invocation Framework (WSIF). Once the reply is ready, the proxy serializes the reply, puts the serialized message into the `genericOutputMessage` (lines 8-10), and sends the message back to the adapted BPEL process. When the `genericOutputMessage` arrives at the adapted BPEL process, it needs to be deserialized. Since BPEL is not a general-purpose programming language, it lacks the necessary constructs to deserialize the (`genericOutputMessage`) message. So, the adapt-ready BPEL process uses the `extract` operation of the generic proxy to accomplish this task. We note that the serialized output message can be comprised of one or more parts [15]; therefore, a series of calls to the `extract` operation might be required to extract the value for each message part. The `extractInputMessage` (lines 12-14) for the `extract` operation has two parts: `values` and `param`. The `values` part contains the serialized

```
1.  <message name="genericInputMessage">
2.    <part name="invokename" type="xsd:string"/>
3.    <part name="porttype" type="xsd:string"/>
4.    <part name="operation" type="xsd:string"/>
5.    <part name="variables" type="xsd:string"/>
6.  </message>
7.
8.  <message name="genericOutputMessage">
9.    <part name="reply" type="xsd:string"/>
10. </message>
11.
12. <message name="extractInputMessage">
13.   <part name="values" type="xsd:string"/>
14.   <part name="param" type="xsd:string"/>
15. </message>
16.
17. <message name="extractReply">
18.   <part name="value" type="xsd:string"/>
19. </message>
20.
21. <portType name="proxyPT">
22.   <operation name="genericInvoc">
23.     <input message="tns:genericInputMessage"/>
24.     <output message="tns:genericOutputMessage"/>
25.   </operation>
26.   <operation name="extract">
27.     <intput message="tns:extractInputMessage"/>
28.     <output message="tns:extractReply"/>
29.   </operation>
30. </portType>
```

Fig. 12. A section of the WSDL description of the interface of the generic proxy.

`genericOutputMessage` and the `param` part specifies which parameter value to extract from the `genericOutputMessage`. The proxy then sends this value back to the BPEL process.

## VIII. Case Studies

In this section, we use three case studies to demonstrate the self-healing and self-optimization behavior of the generated BPEL processes and their respective dynamic proxies. To better demonstrate the applicability of our approach to any BPEL process, we tried to use existing BPEL processes that are not originated by us. 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.

### *A. 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.

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.

*1) The Experiment:* As illustrated in Figure 13, 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".[2]

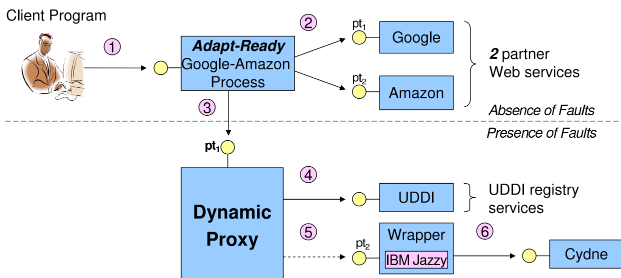



Fig. 13. The sequence of interactions among the components in the Google-Amazon case study.

## B. 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 assessor (simulated) service are deployed locally.

*1) The Experiment:* As illustrated in Figure 14, 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).

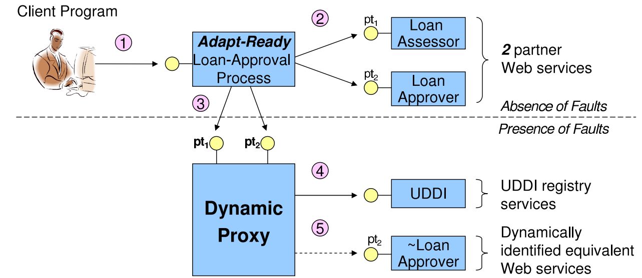



Fig. 14. The sequence of interactions among the components in the Loan Approval case study.

*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 15. 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. 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. We set the timeout duration for the Loan Approval BPEL process to 1 second.

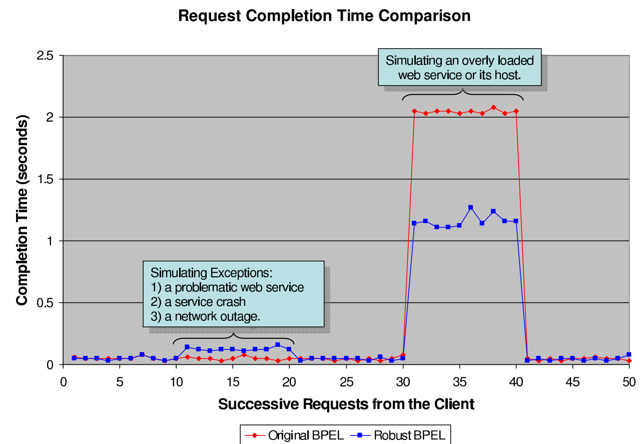



Fig. 15. This chart shows the comparison between the request completion time for the original and the robust BPEL processes.

Figure 15 plots the request completion time for the two sets of experiments (one with the original BPEL process and the other with the generated BPEL process) for the 50 successive requests from the client program

[2]The Amazon store service actually returned a list of books but we only show the first one.

to the Loan Approval BPEL process. One plot reflects the 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 15 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 to find an equivalent service. In its turn, the dynamic proxy uses the UDDI server and finds the substituent service. The plot for robust BPEL in Figure 15 shows an increase in the completion time, which is about 127 milliseconds. We believe that 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 behavior and responds to the requests without throwing exceptions. As can be seen from Figure 15, 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.

### C. The Google-Amazon and Loan Approval Processes

In this part, we evaluate the performance of generic proxies using the two examples previously introduced. As illustrated in Figure 16, for the Google-Amazon process, we have selected the Google spell checker service and for the Loan Approver process, we have selected the Loan Approver service to become adaptable. As can be followed in the figure, a sample policy can be used to forward the intercepted calls to their original services (labels 1, 2, and 3) and use a substitute service in case an original service fails (labels 4 and 5 for the Loan Approval process).

To evaluate the performance hit of the generic proxies, we configured the client applications to sequentially make calls on their corresponding processes. As the X-axis

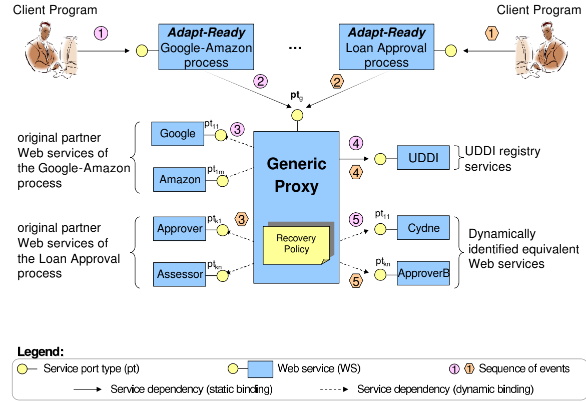



Fig. 16. The interaction between the processes and the generic proxy.

of both charts in Figure 17 shows, the number of total consequent calls are 50. The initial runs were made against the original BPEL process. The results are plotted in the charts in Figure 17 under the *original* curves. Similarly, the results of the observed completion times of the adapt-ready versions are plotted in Figure 17 under the *adapted* curves.

For the Loan-Approval process, the average completion time for the original process is approximately 0.06 seconds and for its adapt-ready version is 0.11 seconds. For the Google-Amazon process, the average completion time for the original process is approximately 0.82 seconds and for its adapt-ready version is 0.86 seconds. This experiment shows that the performance overhead introduced by the generic proxies (approximately 0.05 and 0.04 seconds) is negligible for most BPEL processes.

Figure 18 show a comparison of the average completions times for the static, dynamic and generic proxies on running the adapt-ready Loan-Approval process. The average completion time for the static proxy is 0.078, while that of the dynamic proxy is 0.1, as compared to 0.11 for the generic proxy. We attribute to noticeably lower time for the static proxy to the fact that the substitute service is statically bound to the proxy, which eliminates the overhead incurred through service discovery.

## IX. Related Work

Birman et al. [6] propose 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). Similar to ours, this work aims to improve the reliability of Web services, but it proposes extensions to the Web services architecture.

Baresi's approach [37] 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 achieves the desired separation of concern; however, it requires modifying the original BPEL processes *manually* and the

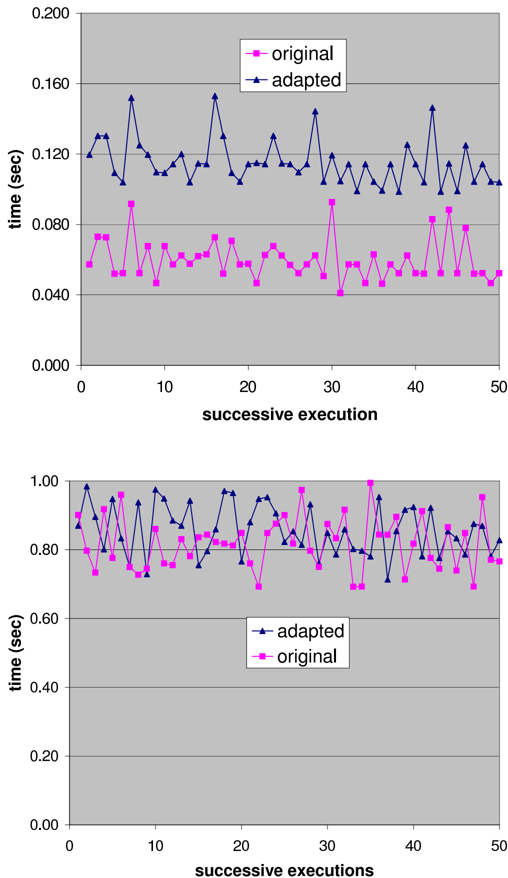



Fig. 17. The response time of the original and adapt-ready processes (in seconds). Top: Loan-Approval Process; Bottom: Google-Amazon Process.

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 [32] use an aspect-based container to provide middleware support for BPEL. The two inputs to the framework are the BPEL process and a deployment descriptor. The descriptor specifies the non-functional requirements (*e.g.*, security, persistence and transactions). 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 can be generated using the deployment descriptor to specify the pointcuts. 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).

Finally, Erradi et al. [38] provide reliability through a policy driven middleware named 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 (*e.g.*, retry, skip, and use equivalent service). The wsBus

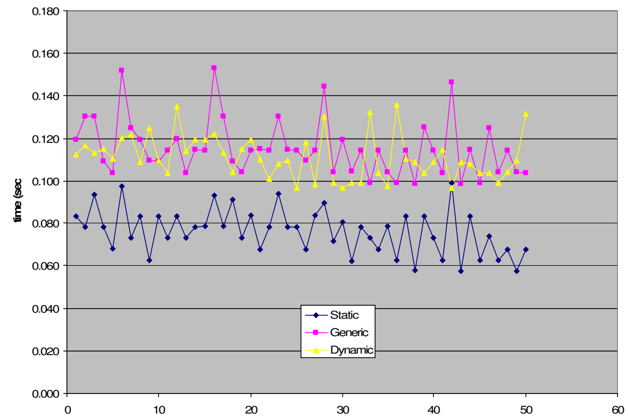



Fig. 18. This charts the average completion times for the static, dynamic and generic proxies.

provides exception-handling and recovers from failures such as service unavailability and timeout. It 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.

## X. 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 static, dynamic, and generic proxies and discussed how they are different and depending on the application which one may perform better than the others. With the use of case studies, we demonstrated how our TRAP/BPEL framework enables us to incorporated self-healing and self-optimization behavior into existing composite Web services developed in BPEL. We use a generative approach to incorporating the autonomic behavior in existing BPEL processes. The adaptation is done in a manner that the code for business logic is kept separated from the code for robustness of the process.

## References

[1] D. Booth et al. , *Web Services Architecture*, W3C, 2004.
[2] S. Weerawarana and F. Curbera, “Business process with bpel4ws: Understanding,” *Online article*, 2002.
[3] D. Sherman, “Business flows with bpel4ws,” *Online article*, 2005.
[4] D. C. Marinescu, *Internet-Based Workflow Management: Towards a Semantic Web*. Wiley-Interscience, 2002.
[5] C. Peltz, “Web services orchestration a review of emerging technologies, tools and standards,” *Technical Paper*, January 2003.
[6] 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)*. Edinburgh, United Kingdom: IEEE Computer Society, May 2004, pp. 17–26.

[7] 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)*, ser. Lecture Notes in Computer Science. Padeborn, Germany: Springer-Verlag, aug 2002.

[8] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *IEEE Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[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] H. Kreger, *Web Services Conceptual Architecture (WSCA 1.0)*, IBM Software Group, May 2001, available at URL: http://www-306.ibm.com/software/solutions/webservices/pdf/WSCA.pdf.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, ser. Addison-Wesley Professional Computing Series. New York, NY: Addison-Wesley Publishing Company, 1995.

[12] 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.

[13] O. Ezenwoye and S. M. Sadjadi, "RobustBPEL2: Transparent autonomization in business processes through dynamic proxies," in *Proceedings of the 8th International Symposium on Autonomous Decentralized Systems (ISADS 2007)*, Sedona, Arizona, March 2007.

[14] O. Ezenwoye and S. M. Sadjadi, "TRAP/BPEL: A framework for dynamic adaptation of composite services," in *Proceedings of the International Conference on Web Information Systems and Technologies (WEBIST 2007)*, Barcelona, Spain, March 2007.

[15] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, *Web Services Description Language (WSDL) 1.1*, 1st ed., W3C, March 2001, available at URL: http://www.w3c.org/TR/wsdl.

[16] K. D. Gottschalk, S. Graham, H. Kreger, and J. Snell, "Introduction to web services architecture." *IBM Systems Journal*, vol. 41, no. 2, pp. 170–177, 2002.

[17] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, and H. F. Nielsen, *SOAP Version 1.2*, 1st ed., W3C, 2003.

[18] T. Andrews et al., *Business Process Execution Language for Web Services version 1.1*, 1st ed., BEA Systems, International Business Machines Corporation, Microsoft Corporation, SAP AG, and Siebel Systems., May 2003.

[19] 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 1st Workshop on the Design and Evolution of Autonomic Application Software 2005*, St. Louis, Missouri, May 2005.

[20] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng, "Composing adaptive software," *IEEE Computer*, pp. 56–64, July 2004. [Online]. Available: http://csdl2.computer.org/dl/mags/co/2004/07/r7056.htm

[21] V. P. Nelson, "Fault-tolerant computing: Fundamental concepts." *IEEE Computer*, vol. 23, no. 7, pp. 19–25, 1990.

[22] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 01, no. 1, pp. 11–33, 2004.

[23] "Software fault tolerance: Fundamental concepts," http://wwwse.inf.tu-dresden.de/presentations/SFT2004-lec1b.pdf.

[24] J.-C. Laprie, "Dependable computing and fault tolerance: Concepts and terminology," in *25th International Symposium on Fault-Tolerant Computing (FCTS-25)*, vol. 3, Pasadena, California, June 1995.

[25] A. Erradi, P. Maheshwari, and V. Tosic, "A policy-based middleware for enhancing web services reliability using recovery policies," in *Proceedings of the 2006 IEEE International Conference on Web Services*, Chicago, USA, September 2006.

[26] S. Tai, T. Mikalsen, and I. Rouvellou, "Using message-oriented middleware for reliable web services messaging." in *Second International Workshop of Web Services, E-Business, and the Semantic Web*, 2003, pp. 89–104.

[27] S. Goel, H. Sharda, and D. Taniar, "Message-oriented-middleware in a distributed environment." in *Third International Workshop on Innovative Internet Community Systems*, June 2003, pp. 93–103.

[28] "HTTPR specification," http://www.ibm.com/developerworks/library/ws-httprspec/.

[29] L. Gilman and R. Schreiber, *Distributed Computing with IBM MQSeries*. Wiley, 1996.

[30] "WS-Reliability 1.1," November 2004, http://docs.oasis-open.org/wsrm/ws-reliability/v1.1/wsrm-ws_reliability-1.1-spec-os.pdf.

[31] S. Tai, T. Mikalsen, E. Wohlstadter, N. Desai, and I. Rouvellou, "Transaction policies for service-oriented computing," *Data and Knowledge Engineering*, vol. 51, no. 1, pp. 59–79, 2004.

[32] 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.

[33] J. Yu and R. Buyya, "A taxonomy of scientific workflow systems for grid computing," *SIGMOD Record*, vol. 34, no. 3, pp. 44–49, 2005.

[34] J. Yu and R. Buyya, "A taxonomy of workflow management systems for grid computing," *Journal of Grid Computing*, vol. 3, no. 3-4, pp. 171–200, September 2005.

[35] 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)*, Seattle, Washington, June 2005, pp. 88–95.

[36] A. T. Manes, "Registering a Web service in UDDI," http://webservices.sys-con.com/read/39868.htm.

[37] 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*. ACM Press, 2004, pp. 193–202.

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


**Onyeka Ezenwoye** holds a Bachelor's degree in Software Engineering from University of Manchester in England and a Master's degree in Computer Sciences from Florida International University, Miami, where he is currently a doctoral candidate. His research is in adaptive software with a current focus on transparent adaptive middleware for application integration. He is also interested in and has worked on problems in software engineering, service-oriented architectures, autonomic computing and Grid computing.

**S. Masoud Sadjadi** received the BS degree in Hardware Engineering from University of Tehran in 1995, the MS degree in Software Engineering from Azad University of Tehran in 1999, and the PhD degree in Computer Science from Michigan State University in 2004. He is currently an assistant professor in the School of Computing and Information Sciences at Florida International University. He has extensive experience in software development and leading large scale software projects. Currently, he is leading several international research projects in the Latin American Grid and is co-chair of the program committee for IEEE ICNSC 2008. His current research interests include Software Engineering, Distributed Systems, and High-Performance Computing with the focus on Autonomic, Pervasive, and Grid Computing. He is PI or Co-PI of 8 grants from NSF and IBM for total of over $3.5 million. He is a member of the IEEE.