Resilient Web Services for Timeless Business Processes

Tomasz Miksa
Rudolf Mayer
Marco Unterberger
SBA Research
Vienna, Austria
{tmiksa, rmayer,
munterberger}@sba-research.org

Andreas Rauber
Vienna University of Technology
& SBA Research
Vienna, Austria
rauber@ifs.tuwien.ac.at

ABSTRACT

Many business and scientific processes make extensive use of service-oriented architectures, using distributed services. These are often provided by third parties and are thus not under direct control of process owners. In this paper we discuss the issues of ensuring continuous and faithful execution of processes in distributed environments, focusing specifically on Web Services. Recently, we introduced a specification of Resilient Web Services, that makes current Web Services more robust, and a framework for the monitoring of Web Services, that allows detecting anomalies. In this paper, we describe alternative implementations of the framework for monitoring of Web Services. We also present possible approaches easing the deployment of Resilient Web Services: a framework consisting of tools deployable at the Web Service operator site enabling easy transformation of a regular Web Service into a Resilient Web Service, and a registry with notifications that decorates existing Web Services with resilient methods.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.4.7 [**Organization and Design**]: Distributed systems

General Terms

Design, Performance

Keywords

Resilient Web Services, SOA, Monitoring, WSMF, Business Continuity

iiWAS '14, December 04 - 06 2014, Hanoi, Viet Nam
Copyright 2014 ACM 978-1-4503-3001-5/14/12...$15.00
http://dx.doi.org/10.1145/2684200.2684281.

1. INTRODUCTION

Rapidly changing business requirements lead to a change of IT systems' design, and resulted in a shift from a centralized designs towards modular and distributed approaches [15]. Thus Service Oriented Architecture (SOA) emerged and became a firm foundation for cloud computing. Business processes deployed in SOA are implemented using discrete, loosely-coupled services that interoperate using a network and have clearly defined communication interfaces. Web Services (WS) are a common way to realize SOA services. Due to the fact that the Web Services can be reused and shared by different business processes, the time needed for implementation of new business requirements has decreased [18]. Besides providing access to the internal resources of the organization, Web Services are used to access external resources hosted and maintained by external business partners, for example, in order to exchange information about customers, bookings, financial data, and so on.

The popularity of Web Services and Service Oriented Architecture is not limited to the business domain, it is also common in the domain of scientific research, especially in e-Science [9], where scientists cooperate by exchanging facilities and sharing information in order to look for solutions of important problems of society. Their research is highly distributed, and this is reflected by the underlying IT infrastructure. Web Services are a way to encapsulate functionalities and provide clear interface for provision of data to other researchers.

Despite the unarguable benefits that Web Services bring to business and scientific processes, they can also be a new source of risks that may affect correct execution and longevity of processes. In [14], we identified four major kinds of disruptions: *a*) unavailability of the Web Service hosted by a third-party, *b*) change of the Web Service's communication interface, *c*) change of functionality while the interface stays the same, *d*) behavioural change of non-functional characteristics. Some of these disruptions can be detected in a straightforward manner. For example, when the Web Service becomes unavailable, the process execution stops. However, a Web Service that returns even slightly altered results than expected, might pass by undetected, and can jeopardise the correctness of reasoning made in a scientific experiment, as well as can cause a threat to business continuity.

Failure to provide business continuity can result in significant financial losses or even in a loss of business. Business Continuity Management (BCM) is often used to discover potential risks and introduce controls that can minimize the negative impact of hazardous events on managed business

processes. In case of business processes relying on Web Services, the process operator may not be the owner of all of them. In situations when the correctness of process execution has to be proven because of liability cases, Service Level Agreements or other legal contracts, it is of particular importance to business operators to be able to detect any alterations in process execution as soon as possible. For this reason the BCM plans must also consider the threats posed by SOA architecture. Therefore, tools that monitor correct execution of Web Services and new requirements for Web Service implementation are needed.

This paper discusses the issues of ensuring continuous faithful execution of processes in highly distributed environments which use Web Services to perform tasks. In [14] we discussed the possible changes in Web Services and their potential consequences to business processes. We proposed the Web Service Monitoring Framework that facilitates detection of changes in Web Services. We also defined foundations of Resilient Web Services (RWSs) which are an extension to the current specification of Web Services making them more robust. This paper extends previous work by a description of alternative implementation for WSMF framework using proxy mechanism and specifies in detail the resilient methods of RWS. Furthermore, we describe two alternate approaches enabling implementation of Resilient Web Services, namely: a Resilient Web Service Framework which is a suite of tools deployable at the Web Service operator site enabling easy transformation of a regular Web Service into a Resilient Web Service, and a register with notifications that decorates existing Web Services with resilient methods using a third party service. Although the discussion is focused particularly on Web Services, the concepts and solutions presented in this paper can in principle be applied to other kinds of distributed services.

The paper is organized as follows. Related work is discussed in Section 2. The requirements and the specification of the Resilient Web Services is provided in Section 4. The alternative implementations of RWS are discussed in Section 5. Section 6 discusses possible usage scenarios for the Resilient Web Services. Finally, conclusions and future work are provided in Section 7.

2. RELATED WORK

Although SOA facilitates the quick adaptation to changes, it can itself affect the business process directly. These can happen when changes in the underlying Information and Communication Technology (ICT) systems appear. In this section we discuss what kinds of such changes can happen in Web Services. Then we discuss how these changes can be monitored and detected. We also discuss available extensions to Web Services that aim at mitigating the consequences of changes.

2.1 Changes in Web Services

In [14] we divided the ICT changes into two categories: internal and external. The internal changes are all these alterations that are under control of the process owner. This means that the effects of any software or hardware modifications, for example, installation of updates or upgrades in the hardware stack, can be traced and the impact on the correctness of process execution can be evaluated. Furthermore, the internal changes can be planned in advance and any possible risks can be identified and corresponding controls mitigating them can be assigned. The external changes are all these modifications that are beyond direct control of the process owner. This is a case when the services are hosted by third parties, for example, using Cloud provider. Then it may happen that the administrator modifies the software or hardware environment, or the system automatically allocates different resources than usually and the business task is affected. The changes can also be further broken down into four categories that differentiate ways the Web Services have changed. Table 1 presents overview of them.

2.2 Monitoring

The goal of Business Activity Monitoring (BAM) is to provide real time information about the status and results of various tasks and processes, thus enabling the management to make better business decisions and quickly address detected problems and opportunities [13]. However, the focus of BAM tools is on monitoring and analysing the processes in view of maximizing revenue and modelling business needs. They are powerful tools for managers, but are not able to detect directly any change of execution stemming from changes in the underlying ICT infrastructure. Only if proper execution of processes is monitored at the technical level, a trustworthy business process execution can be guaranteed. Below we present an overview of solutions capable of monitoring, testing or validating the Web Services.

The first group of papers have the WS-Business Process Execution Language (BPEL) language as a common denominator. The framework presented in [4] generates and executes automatically "online" tests for conformance testing of a composite of Web Services described in BPEL. This "online" approach was combined with passive testing, that verifies time traces with respect to a set of constraints in [3]. Both papers are limited to Web Services which are implemented according to BPEL specification. When the specification is unavailable, the methods cannot be applied. Verification of behavioural conformance of services during runtime is presented in [6]. An idea to apply Stream X-machines in order to check the control flow of a Web Service and the values of the data in the generated responses is shown. The traffic is intercepted from a live system and continuous monitoring for changes is performed. Manual development of an Stream X-machine is required. The other implies access to the Web Service specification which limits the application of this method. A classification of Web Services is also carried out in this paper. Three major criteria are distinguished: conversational / non-conversational, private-state / shared-state, transient-state / persistent-state. In the remaining of this paper we consider these criteria as sub-criteria of the stateful / stateless criterion.

The WS-TAXI framework, which seems to have higher applicability during Web Service development and testing rather than for monitoring of already deployed SOA solutions, is presented in [1]. It combines the coverage of Web Service operations with data-driven test generation. It is able to deliver a complete suite of test messages ready for execution, which were generated using a WSDL file. The WS-TAXI generates and uses purely synthetic data which may be quite different from the data exchanged in a process.

Monitoring if Service Level Agreements (SLA) conditions are fulfilled by Web Services is a problem related to monitoring Web Services for changes. In [8] a run-time monitoring

Change type	Description
Unavailability	Likely stops the execution of the process. The reasons can range from temporary technical problems, to bankruptcy of the service provider. It can be easily detected, for example by using time-outs which would alert to unavailability of the Web Service.
Interface change	Such situation may also be easily detected. It may require short pauses in the process execution until the changes will be adopted into the process. Of course, in case of significant changes in the communication interface (e.g. switch from REST to WSDL), time needed for reconnecting the Web Service into the process may require more effort.
Functionality change	Outputs of the Web Service change, while the interface stays the same. This threat is hard to detect, as the process may not break, but instead deliver outputs which are not correct. These could be, for example, changes at the semantic level, e.g. switching the unit of measurement from inches to centimetre due to a server configuration change. Other possibilities are bug fixes in the underlying algorithm (which may introduce other bugs as well), or intentional changes in the functionality, e.g. faster but less accurate computational algorithms.
Behavioural change	It may not always refrain the process from correct execution, but can occur temporally and therefore be hard to notice. The examples of such cases could be different timing characteristics or delays, effects of buffering, and so on.

Table 1: Summary of changes in Web Services [14]

framework which allows concurrently accessing exchanged messages and comparison of them against designed scenarios was designed and developed. The work focuses on QoS aspects, an example of time-out mechanism detecting unavailability of the service is given. It is required to model the monitored Web Service in Orc language. In case of [7] emphasise is put on detection of violations at functional level. SLAs are described formally using temporal logic and are used in SLAMonitor to verify the behaviour of Web Services at runtime. The authors demonstrate the capabilities of their solution on an example of a detection of violation of maximum response time.

The common problem of all mentioned solutions is the fact that they demand some specific knowledge: not only the kind and the nature of the Web Service, but also the kind of change which will be monitored, is required to deploy a proper solution. In typical situations, however, only the URL and interface of the service are known. There might be no information on whether the Web Service is conversational, stateful, deterministic, etc. This kind of knowledge is, however, required to apply the correct tool. The Web Service Monitoring Framework (WSMF) presented in [14] does not hold these requirements and limitations. It allows investigation of any kind of Web Service, and facilitates reasoning about the kind and nature of a service. Then, if the Web Service is deterministic, the monitoring process can be launched and all four types of changes (see Table 1) can be detected. In case of the Web Service being non-deterministic, the monitoring framework is not able to detect any functional changes, but the other three types of change can still be monitored. Details on the monitoring framework are described in Section 3.

2.3 Web Service Extensions

Apart from monitoring of technological level of business processes, many authors postulate a set of improvements in the specification of Web Services, which should lead to a higher sustainability of processes, as well as reduction of the need for continuous monitoring. Below we present an overview of such solutions.

The Universal Description Discovery and Integration (UDDI) is a registry which holds information on registered Web Services. However, neither the registration of the service is necessary, nor does the registry contain sufficient additional information on the service, which would allow the user to obtain information on the kind, nature, behaviour, quality, etc. There are some attempts to enrich the purely functional description of Web Services (bindings, ports, etc.) with performance aspects, namely Quality of Service (QoS). These focus mainly on timing aspects, availability, reputation [5] and pricing [12]. Most work is dedicated to the creation of frameworks which enable detection of Web Services with different QoS [17], rather than solutions which allow to specify explicitly the common qualities for every service. [19] specifies requirements for QoS for Web Services. It lists 13 points which should be fulfilled, but none of them concerns guaranteeing continuity or non-modifiability.

Another approach is represented by works dealing with versioning of Web Services. Yet in this case, approaches do not aim at specifying a way to interweave versioning into Web Service specification, but present workarounds to deal with the currently underspecified Web Service standards [11]. One of the exceptions to this rule is [10], which provides functional requirements for a registry which notifies clients when a version of an interface changes. [11] is a good example of the current common view on versioning. Versioning is understood as a change of interface. Yet, changes in functionality while the interface stays the same are not considered. This is an obvious deficiency. Some best practices are described in [2], but they are predominantly a kind of workaround for only one of the problems, rather than a holistic solution. Summing up, there is a wide range of approaches that deal with assessment of non-functional aspects of Web Services. In most cases, the need for these solutions arises from the fact that current specification of Web Services is not sufficient. In [14] we introduced a concept of Resilient Web Services which also aims at extending specification of Web Services. They are discussed in detail in Section 4.

3. WEB SERVICE MONITORING FRAMEWORK

The Web Service Monitoring Framework, introduced in [14],

can be employed to monitor services and subsequently detect changes. It consists of the following four steps:

1. *Capture* - the communication to and from the the service is intercepted and stored.
2. *Transform* - requests and responses are grouped and enriched with additional meta-data.
3. *Reason* - data is analysed and a type of the Web Service is determined.
4. *Monitor* - requests collected in the Capture step are replayed and the responses obtained are compared against those captured in the first step.

The following situations can occur during the Monitor step. If no responses are received, this may mean that the Web Service is not available: a change in availability occurred, or a change in the interface caused the unavailability. If only some of the messages are missing, then we can assume that the service is available but only a part of the interface has changed. When the service is deterministic and responses do not match the ground truth, then it indicates a change in functionality. If the service is non-deterministic, changes in functionality cannot be detected easily. If timestamps of recorded messages are stored and time intervals between request and response are calculated, then a change in response timing behaviour can be detected regardless of determinism. The time period required to detect changes is mainly driven by the interval of checks defined in the monitoring schedule.

A crucial requirement for using the approach described above is that the Web Service it is applied to does not cause any changes on the world outside the system observed. In situations where this is not the case, e.g. credit card payment transaction systems, such replaying of messages for monitoring purposes cannot be employed. Thus, while not universally applicable, the approach is still useful for a majority of situations, specifically in e-Science settings, where Web Services are deployed primarily for information transformation, collection or computational services.

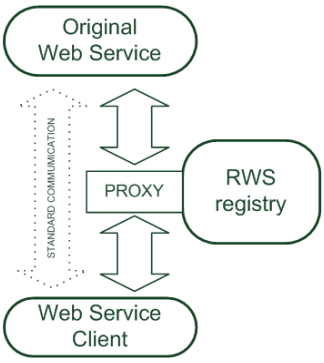

Figure 1: Communication between parties involved using the WSMF framework in a proxy mode.

If the communication can not be directly intercepted at the service consumer or provider side, an alternative implementation is to use a proxy mode. In this case, the communication between the consumer and the provider is temporarily redirected through a proxy server, which captures the requests, and forwards them to the actual server; subsequently, the server responses are relayed to the consumer. Thus, the exchange of real requests and responses can be intercepted. Figure 1 depict the communication paths between the involved parties in the proxy mode. Figure 2 shows the proof of concept tool that implements the Web Service Monitoring Framework. The figure depicts a situation in which three requests are tested against the originally recorded results, and their response validity is evaluated. The implementation also includes the proxy mode.

4. RESILIENT WEB SERVICES

In this section we discuss Resilient Web Services as an extension to the WSDL Web Service specification. We identify and describe methods that are required in order to transform a regular Web Service into a Resilient Web Service.

4.1 Motivation and design principles

The main aim of the resilient methods is to help the WS consumer to react to changes within the service. The long-term sustainability and usage of WSs would positively impact the longevity of business processes depending on them. Having analysed the existing monitoring and extension mechanisms, we defined principles of RWS design.

4.1.1 Minimal availability date

It is very common, especially in the scientific domain, that when the WS consumers select WSs, which are used in their processes, they have no information and guarantee on how long the given WS is going to be available and thus how sustainable their process is. The RWSs provide minimum availability date of the WS. This information is a commitment from the WS owner, and states how long the WS is going to be present on-line in an unchanged form. The only exceptions to this agreement are the alterations caused by *force majeure*. The services that specify this date could be considered more trustworthy than those without such a commitment. It also implies that the owner of the WS is providing rather a stable service that has secured funding and resources for the maintenance of the WS. The WS consumer knowing in advance when the WS may become unavailable in the current form may prepare for this. They can either contact the WS owner and discuss the changes, so that the business process is adapted to changes, or may look for a different WS or another solution that delivers the functionality of a given WS.

4.1.2 Identification data

Many of the available WS are not registered in services like the UDDI and therefore it is hard to obtain any information going beyond the technical specification described in the WSDL. Sometimes the complementary descriptions are published on the web sites of the WS owners. However, in cases when the WS owner runs a wide range of similar services (e.g. ranging in precision of computing algorithms) then it may be not obvious which description applies to which WS. For this reason, we believe that provision of basic information about the WS, which is obtained using the WS methods, can highly ease the WS discovery process. The essential information which must be provided is: version, description, type of methods, and metrics related to the Quality of Service. The version number allows unambiguous identification of the WS. The description should summarize the function-

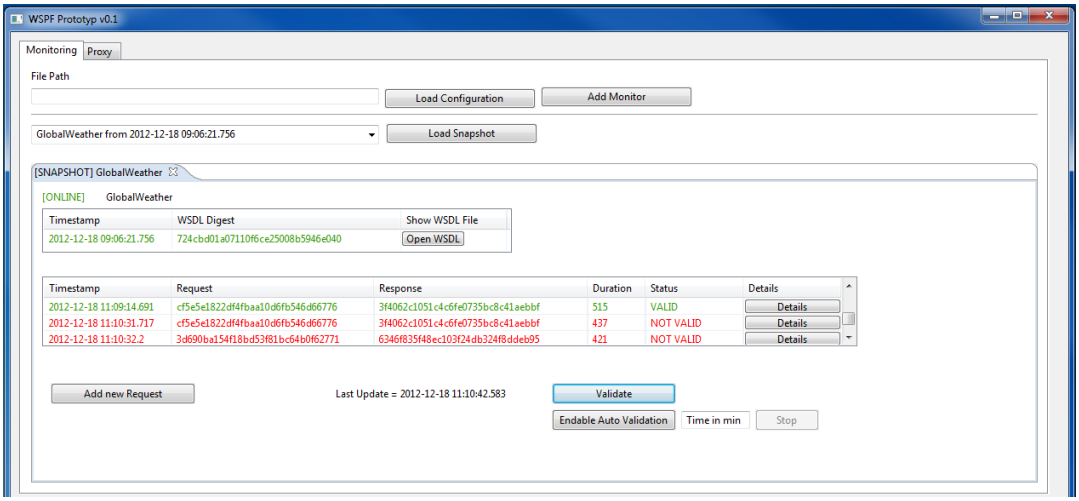

Figure 2: Software Support for the Web Service monitoring framework, including a proxy mode

ality of the WS and be kind of the release notes, that is, describe the differences towards the previous versions. The information about the method type is essential for monitoring and testing of WSs. At least information on whether the method is stateful or stateless, and whether it is deterministic or non-deterministic, has to be provided. Also a list of metrics related to the Quality of Service, for example, the date of the last update or a number of changes to the WS since its first deployment should be provided. Such information helps in assessment of the stability of the WS, and is a good premise for reasoning about the future frequency of changes of the WS.

4.1.3 *Documenting Changes in the SW/HW stack*

Web Services usually encapsulate complex processes which depend on various software tools, libraries, codecs, databases, or virtual machines, not to mention the hardware platforms and special instruments used, for example, in scientific research. For this reason, any change in one of these components may result in a different behaviour of the service, including the final result it computes. Of course, not every change within the underlying system has the impact on the WS, however, in cases when there is such an impact, the list of software or hardware components that have been modified needs to be provided. For example, if a new sensor type is used to take measurements, then the information on the sensor introduced must be provided. Similarly, if a software library was replaced with another library that implements a different computational model, then this information must also be provided. Currently, there is no mechanism that provides such information and hence unexpected changes in the WSs occur. For that reason a common way of providing information on changes in software or hardware that constitutes the system behind the WS is needed. This information helps the WS consumer to estimate the impact of the technological change on the performance of the whole business process. For example, if the change resulted in more detailed results of some computation and the software at the WS client can process this data (because there was no change in format and no other business conditions are violated), then the WS consumer may continue using a given WS. Obviously, some of the information provided might incur vulnerabilities on a security level. Therefore, the level of details provided should be parameterised on different level depending on the trustworthiness of the environment or authentication levels of the client services.

4.1.4 *Notification on changes*

As already discussed (see Table 1) the changes are inextricably intertwined with WSs. The common belief of the IT community is that the changes are inevitable and instead of suppressing them we need to be able to react to them quickly and therefore design systems and processes that can adapt quickly. The WSs and the SOA architecture were designed having this in mind. However, their aim is to adapt to changes in the business requirements, not to the changes in the underlying infrastructure that implements the SOA and the WSs. Due to this fact, the extension of WSs with a mechanism that provides notification on changes in the service itself complements the WS design with the focus on the infrastructure changes. Hence each RWS has a method that notifies the WS consumers, when any kind of change to the WS happens. The Web Service communication is asymmetrical and this notification can only be received when requested by the WS consumer. Hence, the WS consumers will have to poll for this information, but such a solution still has an advantage over traditional monitoring at the WS consumer side, mainly because it is easier to implement in comparison to a deployment of any of the monitoring frameworks. Furthermore, the user receives not only information that something has changed, but also knows what kind of change it is and also a message from the WS operator explaining it. Further information concerning the change can be requested using the methods that provide a list and description of changes as described in the paragraph above.

4.1.5 *Service owner and contact data*

The importance of the need for contact between the WS consumer and the WS provider was already recognized by the architects of the UDDI registry and is reflected by the business entity data structure. This information is also similar to the *whois* protocol used for querying DNS databases and consists of contact data of the WS owner: name, phone, e-mail address, and so on. However, there are many WSs

```
<xs:element name="response">
  <xs:complexType>
    <xs:sequence maxOccurs="unbounded" minOccurs="1">
          <xs:element type="xs:dateTime" name="changeDate"/>
          <xs:element type="CHANGETYPE" name="changeType"/>
          <xs:element type="xs:string" name="changeDescription" use="optional"/>
          <xs:element type="xs:string" name="changesList"/>
      </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:simpleType name="CHANGETYPE">
    <xs:restriction base="xs:string">
      <xs:enumeration value="Unavailability" />
      <xs:enumeration value="InterfaceChange" />
      <xs:enumeration value="FunctionalityChange" />
    <xs:enumeration value="BehaviouralChange" />
   </xs:restriction>
</xs:simpleType>
```

Figure 3: Fragments of an XSD schema defining format of the response for *getChangesSince* method.

that are not registered in any registry and due to this fact, the contact between the WS consumer and WS owner is impossible. Despite the fact, that other resilient methods aim to describe the WSs comprehensively and provide information about the changes, thus decreasing the need of direct communication between the parties, it may happen that the direct contact may considerably facilitate dealing with changes and can enable support from the WS owner to the WS consumer. For this reason, the RWSs have a method that provides contact data of the WS owner. We believe that a standardised method for obtaining this information will become more popular than the centralized registry like the UDDI. This is because the registration at the UDDI was optional and in some cases cumbersome. Furthermore, similarly to the methods dealing with identification of the WS, we believe that the provision of the contact data directly by the RWS results in less ambiguities concerning not only the ownership of the WS, but also matching the description with a particular service.

4.2 Specification

Using the requirements from the previous section we elaborated a list of methods that constitute RWS. In this section we specify the methods and provide examples of responses obtained from these methods.

minAvailabilityDate() This method has no input parameters and in its output provides the guaranteed minimum availability date of the WS. In other words it provides the deadline till which the given WS is going to perform in an unchanged way. This method addresses the requirements defined in Section 4.1.1.

identifyYourself() This method address the requirements defined in Section 4.1.2. This method has no input parameters. The output consists of multiple information:

- version - number indicating release version of the WS, the version number is updated every time the WS owner modifies intentionally the WS,
- description - textual description of the latest release of the WS, it should contain information on differences towards the previous version,
- methods and their types - a list of all methods offered by the WS and information on its type, there are four available types: StatelessDeterministic, StatlessNonDeterministic, StatfulDeterministic, StatefulNonDeterministic,
- first release data - date since when the WS is available,
- total number of changes - integer indicating how many changes to the web service have been detected since its beginning, it includes both expected (new versions) and unexpected changes,
- availability percentage - availability of the WS expressed in percentage.

getSystemEnv() This method has no input parameters. It addresses requirements provided in the Section 4.1.3. The output is an ontology describing software and hardware components which are crucial for functioning of the WS. To describe the system environment in a structured way, we employ a meta-model that is well-known and widely used in the domain of Digital Preservation, namely the PREMIS Data Dictionary [16]. PREMIS allows to describe various aspects of computing infrastructure, including hardware and software of systems, which is of particular interest for our purpose. The data dictionary defines five types of entities: Intellectual, Object, Event, Agent, and Rights. It then defines 45 concepts belonging to these types, as well as relations between the concepts and properties of the concepts. For describing the system environment, we use the OWL ontology representation of PREMIS[1]. Specifically, the concepts of *Hardware* can be described by name, type, and additional free-text descriptive information, while the concept of *Software* has in addition also a property to specify the software version. A piece of software might require another software to be installed to properly function, which can be described by using the relation *hasSoftwareDependency* between two specific software instances. With these concept, we can sufficiently describe the current hardware and software setup of a specific system. This detailed description is only available to certain actors, namely a web service owner who is providing the service on serviced infrastructure.

[1] `http://id.loc.gov/ontologies/premis.html#`

getChangesSince(DateTime) There is one input parameter to the method which is the exact date and time since which all potential changes are listed in the output. This date could be the date of the last request sent to the RWS by the WS consumer. If no changes were detected since that time, then the result is empty. Otherwise a list of all changes is returned. Each change is described with the following information:

- change date and time - exact date and time of the change,
- change type - type of change as defined in Table 1,
- change description - textual description of the change, it is optional and to be used in cases when the change notification is done manually,
- change list - ontology listing hardware and software components that were modified.

Changes in the system environment can also be described using PREMIS. To this end, the *Event* concept can be utilised. For example, we can indicate a replacement of a software component as an event of type *migration*, with an associated description using the *EventOutcome* concepts. The old and new components are related to the event via source and outcome relations. Figure 4 provides an excerpt from an example response. Figure 3 presents an excerpt of the XSD schema defining the format of the response send by this method. The types of changes are encoded as enumerations and the ontology with a list of changes is provided as a string directly in the body of the response. The method addresses requirements described in the Section 4.1.3 and 4.1.4.

getContact() This method has no input parameters. It addresses the requirements from the Section 4.1.5. The output provides contact details using entities from the UDDI schema. The following information is returned:

- organisation - name of the organisation owning the WS,
- person name - name of the person to be contacted, it is not necessarily identical with the WS owner,
- message - optional message from the contact person,
- phone - optional phone number,
- email - email address used for contacting,
- address - optional address.

5. IMPLEMENTATION

Industry uptake of a new standard can be a slow process. Thus, we propose two approaches to make our proposed Resilient Web Services easier and quicker to be deployed. The first approach is by providing an external registry that monitors the service and its behaviour, and thus can provide some of the resilient methods without any need for modification of the original service. This will be described in detail in Section 5.1. The second approach is in providing tools to the service owner to quickly transform an existing service to a resilient service, by deploying it on top of a framework that already implements most resilient methods; we show this in Section 5.2 with a prototype implemented in Java.

5.1 External Registry

To allow resilient methods to be provided on top of an existing service, without requiring any changes in the service or its deployment, we propose the concept of an external registry, which send notifications to the service consumer. The main task of the registry is the decoration (design pattern) of existing Web Services with resilient methods.The registry is a service provided by a third party. Such an approach should substantially increase the acceptance of Resilient Web Services among service operators and thus considerably decrease the adoption time. In this section, we therefore discuss how such a registry works, what actions are required from the parties involved and in what way it can be implemented.

Figure 5 illustrates the process of converting a WS into a RWS using the registry. There are three actors involved: the Web Service provider, the Web Service consumer, and the registry operator. When using traditional Web Services, the WS provider publishes a WSDL specification of the WS, and the consumer uses it to establish a connection to the WS. In the approach utilising the registry, the communication is still realised directly between the provider and the consumer, but the consumer obtains information about the port bindings from a different WSDL definition that is provided by the registry operator. The WSDL that is obtained from the registry consists of two logical parts. The first one is generated using the original WSDL file of a given Web Service, by verbatim copying information about the Web Service methods. The second one provides bindings to the resilient methods provided by the registry. Thus no changes at the side of WS provider are required. Furthermore, the communication between the provider and consumer remains unaltered and therefore no unnecessary complexity is added.

The data provided by the resilient methods consists of two types of information: static and dynamic. The static information is provided once when a given WS is registered at the registry. This is only possible when the registration is made by the WS provider, because they have the necessary knowledge about the WS. An example of such a static information that can only be provided by the WS owner is the expiration date. The active information comes from the monitoring of the registered Web Service. This functionality is provided regardless of who registered the WS. Both provider and consumer can register the WS, however, it is recommended that the registration is made by the WS owner, because more resilient methods can be used.

The active information provided by resilient methods of the registry come from monitoring that can be implemented using the Web Service monitoring framework described in Section 3. However, the implementation described in [14], which uses network packet capturing in order to collect data used for monitoring, cannot be applied in case of the registry. This is because the registry is provided by a third party and therefore does not have a direct access to the network interface of the service consumer. For this reason two alternative approaches can be used. Either a set of synthetically generated requests can be used, as described in [1], to query the original WS and collect responses, or the proxy mechanisms described in Section 3 can be applied.

We implemented the proposed registry in Java, and made a use of the aforementioned monitoring tool using the proxy mode. We simulated changes on a number of Web Services for which we had access to the source code, to detect all

```
<ClassAssertion>
  <Class IRI="http://id.loc.gov/ontologies/premis.rdf#Event"/>
  <NamedIndividual IRI="[serviceLocation]/[identifier]/SoftwareReplacement"/>
</ClassAssertion>
<ObjectPropertyAssertion>
  <ObjectProperty IRI="http://id.loc.gov/ontologies/premis.rdf#linkingSourceObject"/>
  <NamedIndividual IRI="[serviceLocation]/[identifier]/SoftwareReplacement"/>
  <NamedIndividual IRI="[originalModelURI]#OracleJava1.6.u44"/>
</ObjectPropertyAssertion>
<ObjectPropertyAssertion>
  <ObjectProperty IRI="http://id.loc.gov/ontologies/premis.rdf#linkingOutcomeObject"/>
  <NamedIndividual IRI="[serviceLocation]/[identifier]/SoftwareReplacement"/>
  <NamedIndividual IRI="[serviceLocation]/[modifiedModelURI]#OpenJDK1.7.u65"/>
</ObjectPropertyAssertion>
```

Figure 4: Description detailing the changes made to the system by replacing *Oracle Java* version 1.6 with 1.7

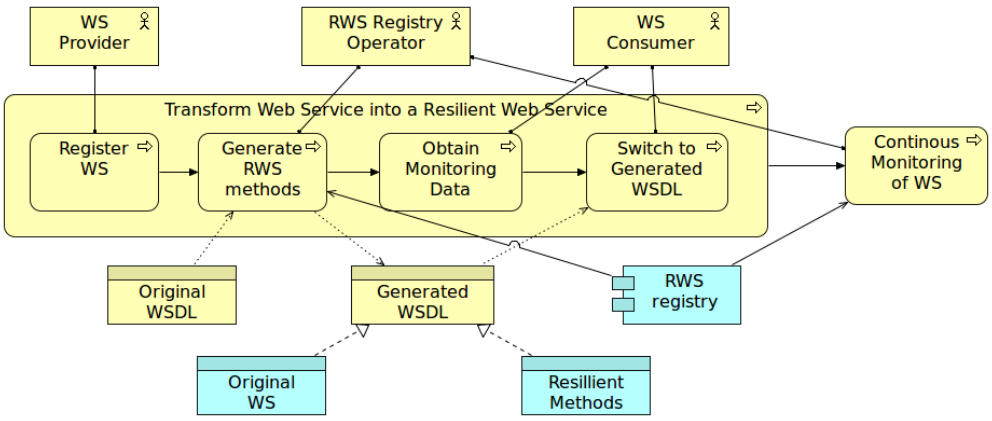

Figure 5: Business process model, depicting the transformation of a Web Service into a Resilient Web Service.

types listed in Table 1.

5.2 On-site Monitoring

In a second approach towards implementing Resilient Web services, we deploy certain components directly on site of the web service provider. Compared to the purely register based approach, this setup allows us to detect potential causes for service changes directly in the environment, for example, changes in the hardware or software setup that might influence the functionality of the Web Service. Thus, we can trigger in a more informed way when we have to run our monitoring again, and if changes occurred, the likely source can be specified.

Most of the resilient methods introduced in Section 4 do not require an implementation specific to a given Web Service. As such, the hardware or software setup of the machine can be determined via methods that are generic regardless of the actual implementation of the Web Service, but can be determined in the same or similar fashion for all Web Services using e.g. the same type of Operating System. Some methods may depend on the specific programming language used for implementing a Web Service, e.g. the type of libraries that can be utilised, and the way they are declared as dependencies, but again, these aspects can be generalised for Web Services using the same type of development platform. Therefore, we can provide an implementation of many of the resilient methods tailored for a specific platform or programming language. These implementations can then be used by the service provider to augment the service to become a Resilient Web Service. As an example, we describe below an implementation of such a Resilient API for the Java Programming language.

Manually providing information on hardware and software, and monitoring for the changes, can be a tedious task. However, many of these steps can be automated, or be semi-automated. Regarding hardware, several tools provide automatic detection of the components accessible to the operating system, such as the platform-independent SIGAR framework[2], or utilities such as *lhw*[3] for Linux. Regarding the software environment, several methods allow identification of the currently installed software, such as via the Windows Management Instrumentation Command-line (WMIC) for Windows systems, or package managers such as the Debian package manager *dpkg* or the RedHat package manger *rpm*. In the course of the EU FP7-funded Project TIMBUS[4], which deals with the digital preservation and continuity of business processes, software modules that periodically capture this information, store it in the PREMIS ontology, and monitor for changes in the environment, were developed. We utilise this software, which is specifically targeted to Linux Operating Systems, at the side of the service provider to im-

[2] `http://www.hyperic.com/products/sigar`
[3] `http://ezix.org/project/wiki/HardwareLiSter`
[4] `http://timbusproject.net/`

Table 2: Support of resilient methods in different deployment scenarios

Method	External registry	On-site Monitoring
identifyYourself()	Y	Y
getContact()	Y	Y
minAvailabilityDate()	Y	Y
getSystemEnv()	N	Y
getChangesSince()	∼	Y
- change type	N	Y
- change description	N	Y
- environment changes list	N	Y

plement the *getSystemEnv* method. Subsequently, once we detect a change in the software or hardware environment, a monitoring of the Web Service as with the registry approach described in Section 5.1 is triggered. Only if there is a change in the Web Service functionality due to the environment change, the *getChangesSince* method will report these changes. In addition to the registry approach, also the description and list of environment changes is provided.

Similar to Section 5.1, we modified a web service environment, by upgrading the Java Virtual Machine from version 6 to version 7 (cf. the example in Figure 4). The on-site monitoring was successfully detecting the change of the environment. This in turn triggered the monitoring of the service functionality, where no change was detected; thus, no notification was sent to the Web Service consumer.

5.3 Client Side Implementation

Once the Web Service has been upgraded into a Resilient Web Service, the service consumer needs to implement support for resilient methods. This implementation can e.g. be similar to the exception handling mechanisms used in programming languages. Due to the fact, that the Web Service communication is asymmetrical, i.e. the provider cannot provide any information without any prior request, the WS consumer must implement a method that regularly polls the resilient method *getChangesSince()* that provides information whether there was a change to the WS, and if so what kind of a change it was. Depending on the kind of change detected, a corresponding scenario can be performed, for example, the execution of the processes using this Web Service can be stopped, or the WS can be substituted with another. Additional information provided by other resilient methods may also be useful in selecting appropriate recovery solution. An overhead resulting from the necessity of these improvements should be acceptable by the WS consumers, because they are the main beneficiaries of the Resilient Web Services. An alternative implementation may use push-style notifications to propagate information on changes, for example, using RSS or e-mail. The advantage of such an implementation is the fact that no polling for changes is required. On the other hand, the client needs to support additional communication protocols (not only SOAP).

6. DISCUSSION

There are several different usage scenarios for the deployment of Resilient Web Services, with different actors and roles involved. On the one hand, there is a potential differentiation on who is registering a service at an external registry. In the ideal case, the service owner is performing this task, but there might also be cases where the service owner is not registering the service himself. In such a case, a service consumer might be allowed to register the service. Such a situation will also imply limited functionality of resilient methods – basically only the *getChangesSince* method is available, as all the meta-data that the service owner would be providing (availability, contact, and so on) is missing.

Another distinction might be if the Web Service provider is hosting the service using services from a third-party, e.g. Infrastructure as a Service (IaaS) or Platform as a Service (PaaS). In such a scenario, the IaaS or PaaS provider is the only one to provide certain information needed for the on-site monitoring and the *getSystemEnv* method, and thus the Web Service provider would become a consumer of this information from the hosting provider.

Web Services are sometimes used to create mashups combining functionality of various Web Services, thus providing a new service. If the Resilient Web Services are used to create such a mashup, then it is possible to forward information on changes from the RWSs used to create it to the mashup. Hence, the mashup is also a Resilient Web Service. In this scenario the mashup owner uses the *getChangesSince()* method to receive notifications on changes from the dependent RWSs. Such a *Resilient Web Service chaining* is possible regardless of the implementation of the RWS. The mashup owner needs to design mechanisms to handle notifications from dependent RWSs and add the resilient methods to the interface of his mashup. The mashup owner does not have to use the registry nor the on-site tools suite, as long as he does not provide any new in-house developed methods that do not depend on RWSs.

Another important aspect, especially in the *getSystemEnv* method, is security. Exposing the exact hardware and software setup to anyone on the Internet might introduce security risks, as potential attack vectors based on vulnerabilities in the hardware, operating system or other software components are easier to identify. Thus, in many scenarios, it might be useful to restrict the information provided by, or the access to the *getSystemEnv* method. Other security-related concerns are also of importance. For example, encryption of the Web Service with one-time keys requires that the monitoring framework can still understand the messages exchanged. Also, tokens or authentication mechanisms that might prevent replaying of messages need to be considered. While these aspects can be easily circumvented with the consent of the parties involved, their commitment that this is desired and allowed needs to be explicit, and the monitoring framework by default is not configured to perform such man-in-the-middle approaches.

7. CONCLUSIONS

In this paper we discussed issues of ensuring continuous and faithful execution of processes in environments that use distributed services to perform tasks. We focused on Web Services and analysed potential changes stemming from them that impact business continuity. The monitoring and testing of Web Services, as well as extension mechanisms enriching the Web Services with additional information on their behaviour and availability were investigated.

Our work put special attention to the recently proposed extension of Web Services, namely the Resilient Web Services. We discussed motivation behind this concept and out-

lined the requirements it fulfils. We also provided a detailed specification of the resilient methods. Furthermore, we described two alternative implementations that should ease its uptake and make the deployment easier and quicker. First, the external RWS registry that allows converting any Web Service into a Resilient Web Service without modification at the WS provider site. For that purpose we had to provide a new implementation of the existing Web Service Monitoring Framework that uses proxy mode to intercept communication. Second, the on-site monitoring tools suite that enables full utilization of resilient methods including information automatically collected from the underlying system. Both of the solutions are capable of providing notification on changes to the WS consumer and thus contribute significantly to the minimization of the impact of changes in the ICT infrastructure on the business processes. Although the discussion in this paper focused on Web Services, we believe that the solutions proposed here can also be applied in other implementations of distributed computing environments.

Currently we are working on identification of basic set of tools that can be used on different platforms to provide ready to use software kit allowing broader community of WS operators to make their services resilient. We are also going to investigate the scalability of the RWS registry and release an open source version of it.

Acknowledgements

This work has been co-funded by COMET K1, FFG - Austrian Research Promotion Agency and by the EU-FP7 funded TIMBUS project (grant agreement no. 269940).

8. REFERENCES

[1] C. Bartolini, A. Bertolino, E. Marchetti, and A. Polini. WS-TAXI: A WSDL-based Testing Tool for Web Services. In *ICST '09. International Conference on Software Testing Verification and Validation*, pages 326 –335, april 2009.

[2] K. Brown and M. Ellis. Best practices for Web services versioning. http://www.ibm.com/developerworks/webservices/library/ws-version/, 2004. Accessed: 30/06/2014.

[3] T.-D. Cao, R. Castanet, P. Felix, and G. Morales. Testing of Web Services: Tools and Experiments. In *Services Computing Conference (APSCC), 2011 IEEE Asia-Pacific*, pages 78 –85, dec. 2011.

[4] T.-D. Cao, P. Felix, R. Castanet, and I. Berrada. Online testing framework for web services. In *Third International Conference on Software Testing, Verification and Validation (ICST)*, pages 363 –372, April 2010.

[5] M. Comuzzi and B. Pernici. A framework for QoS-based Web service contracting. *ACM Trans. Web*, 3(3):10:1–10:52, July 2009.

[6] D. Dranidis, E. Ramollari, and D. Kourtesis. Run-time Verification of Behavioural Conformance for Conversational Web Services. In *7th IEEE European Conference on Web Services (ECOWS)*, pages 139 –147, Nov. 2009.

[7] N. Goel, N. N. Kumar, and R. Shyamasundar. SLA Monitor: A System for Dynamic Monitoring of Adaptive Web Services. In *9th IEEE European Conference on Web Services (ECOWS)*, pages 109 –116, Sept. 2011.

[8] N. Goel and R. Shyamasundar. Automatic Monitoring of SLAs of Web Services. In *Services Computing Conference (APSCC), 2010 IEEE Asia-Pacific*, pages 99 –106, Dec. 2010.

[9] T. Hey, S. Tansley, and K. Tolle, editors. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009.

[10] B. Kalali, P. Alencar, and D. Cowan. A service-oriented monitoring registry. In *Conference of the Centre for Advanced Studies on Collaborative research (CASCON)*, pages 107–121. IBM Press, 2003.

[11] P. Kaminski, H. Müller, and M. Litoiu. A design for adaptive web service evolution. In *Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems*, SEAMS '06, pages 86–92, New York, NY, USA, 2006. ACM.

[12] Y. Liu, A. H. Ngu, and L. Z. Zeng. QoS computation and policing in dynamic web service selection. In *13th International World Wide Web conference*, pages 66–73, New York, NY, USA, 2004. ACM.

[13] D. W. McCoy. *Business Activity Monitoring: Calm Before the Storm*. Gartner Research, 2002.

[14] T. Miksa, R. Mayer, and A. Rauber. Ensuring sustainability of web services dependent processes. *International Journal of Computational Science and Engineering (IJCSE)*, 2015. In press.

[15] A. Mulholland, R. Daniels, and T. Hall. *The Cloud and SOA*. Capgemini, 2008.

[16] PREMIS Editorial Committee. Premis data dictionary for preservation metadata. Technical report, 2008.

[17] M. Tian, A. Gramm, H. Ritter, and J. Schiller. Efficient Selection and Monitoring of QoS-Aware Web Services with the WS-QoS Framework. In *Proceedings of the 2004 IEEE/WIC/ACM International Conference on Web Intelligence*, pages 152–158, Washington, DC, USA, 2004. IEEE.

[18] M. van den Berg, N. Bieberstein, and E. van Ommeren. *SOA for Profit, A Manager's Guide to Success with Service Oriented Architecture*. IBM Press, 2007.

[19] W3C Working Group. QoS for Web Services: Requirements and Possible Approaches. http://www.w3c.or.kr/kr-office/TR/2003/ws-qos/, 2003. Accessed: 30/06/2014.