Raising resilience of web service dependent repository systems

Tomasz Miksa and Rudolf Mayer
SBA Research, Vienna, Austria, and
Andreas Rauber
Vienna University of Technology, Vienna, Austria, and SBA Research, Vienna, Austria

Received 5 April 2015
Revised 21 April 2015
Accepted 21 April 2015

Abstract

Purpose – This paper aims to address the issue of long-term stability of services and systems depending on service-oriented architecture that has become a popular architecture in systems development and is often implemented using Web services. However, the dependency, especially on externally provided services, can impact the reliability of a system. This is often caused by the loose coupling also implying a less stringent policy for change management and notifications. Therefore, the authors characterise the types of changes that can happen in remote services and propose the concept of resilient web services (RWSs) as an example on how to upgrade existing services to better support the long-term stability of services and systems.

Design/methodology/approach – Having analysed several use cases where systems broke because of external dependencies not correctly maintained, the authors derived requirements for RWSs.

Findings – By means of a prototype implementation and evaluation of this solution in a case study, the feasibility of the approach was verified. Several scenarios of changes in WSs were simulated, correctly identified and responded to.

Originality/value – The authors propose a set of extensions to existing standards such as Web Services Description Language to improve the long-term availability of services in SOAs. A prototype implementation was developed for service monitoring and RWSs.

Keywords Context model, Digital repositories, Resilient web services, Service-oriented architecture, WS monitoring

Paper type Research paper

1. Introduction

Service-oriented architecture (SOA) is a popular mean to implement complex systems, especially in, but not limited to, settings where some parts of the system are provided by external parties. Web services (WSs) are a popular pattern to realise SOA. While SOA allows for a rapid development cycle, especially when existing services are recombined as mash-ups, there is also a threat of the reliability and long-term availability of these services, and thus also for applications that build on top of them. These threats can result in an unavailability of the service due to a change in its interface, and also due to a change in functionality of the service, without interface change.

To alleviate these threats and uncertainties, the concept of resilient web services (RWS) has been proposed lately (Miksa *et al.*, 2015, 2014). RWS build on top of existing

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).

International Journal of Web Information Systems
Vol. 11 No. 3, 2015
pp. 327-346
© Emerald Group Publishing Limited
1744-0084
DOI 10.1108/IJWIS-04-2015-0011

WS standards, such as Web Services Description Language (WSDL), by adding a set of methods available to the consumer that facilitate the management and use of the service over time, when changes occur. RWS build on a versioning of WSs and provide a messaging mechanism that informs the service consumers when a change has occurred including information on the nature of the change. The consumers can then verify whether their application is actually affected, and either opt to use an earlier version or adapt their product.

In this paper, we apply the concept of RWS on a real-world scenario from the domain of digital libraries and document repositories. Trust in the long-term availability and reliability of the system is important for digital libraries, as external parties refer to the facts provided by the documents in the system. The system we study in detail is *Phaidra*, which is a system following SOA principles. It is deployed in several local instances among the various academic partners of the development consortium. Some of the service are, however, provided centrally by the lead partner, and are consumed by others. We demonstrate how these services can be converted into RWS, mostly relying on infrastructure and information that is already present. Further, we investigate the required effort from the consumer point of view and evaluate whether the usage of RWS improves the system's reliability.

The paper is organised as follows. Related work is discussed in Section 2. Monitoring of WSs is discussed in Section 3, before the concept of RWS is described in Section 4. Aspects and reference implementations of RWS are discussed in Section 5. Section 6 then introduces the use case of the digital library system, and describes in detail how RWS are provided and consumed in this setting. Finally, a discussion and future work are provided in Sections 7 and 8.

2. Related work

In this section, we discuss what kinds of changes can happen in WSs. Then, we discuss how these changes can be monitored and detected, as well as models that can be used to persist and analyse these changes. We also discuss available extensions to WSs that aim at mitigating the consequences of changes.

2.1 Changes in WSs

In Miksa *et al.* (2015), we divided the information and communications technology (ICT) changes into two categories: internal and external. Internal changes are all alterations that are under control of the process owner. The effects of any software or hardware modifications, for example, installation of updates, can be traced, and the impact on the correctness of process execution can be evaluated. Furthermore, the point in time when internal changes are caused can often be planned in advance, and mitigation actions can be provisioned. External changes are modifications that are beyond direct control of the process owner. For example, when a service is hosted by a third party, it may happen that the administrator modifies the software or hardware environment, or the system automatically allocates different resources than usually. As a consequence, the customer's process is affected.

In addition to the internal/external distinction, changes can further be broken down into four categories that describe the nature of the WS change, as listed in Table I.

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 WS
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 WS into the process may require more effort
Functionality change	Outputs of the WS 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, etc.

Source: Miksa *et al.* (2015)

Table I. Summary of changes in WSs

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 (McCoy, 2002). However, the scope of BAM tools is on monitoring and analysing the processes from a business view, and not on the underlying ICT infrastructure.

The framework presented by Cao *et al.* (2010) automatically generates and executes tests for conformance testing of a composite of WSs described in Business Process Execution Language (BPEL). This approach was combined with passive testing, which verifies time traces with respect to a set of constraints (Cao *et al.*, 2011). Both solutions are limited to WSs that are implemented according to the BPEL specification. Verification of behavioural conformance of services during the run time is presented in the study by Dranidis *et al.* (2009). Stream X-machines are applied to check the control flow of a WS and the generated responses. The traffic is intercepted from a live system, and continuous monitoring for changes is performed. The stream X-machine needs to be developed manually, and requires access to the WS implementation, which limits the application of this method. The authors also provide a classification of WSs. Three major criteria are distinguished: conversational/non-conversational, private state/shared state and transient state/persistent state. In our work, we consider these criteria as sub-criteria of the stateful/stateless criterion.

The WS-TAXI framework (Bartolini *et al.*, 2009) combines the coverage of WS operations with data-driven test generation. It is able to deliver a suite of test messages ready for execution, generated using a WSDL specification. WS-TAXI generates and uses purely synthetic data which may be quite different from the data exchanged in a process. It is thus more suited for WS development and testing, rather than monitoring of the already deployed SOA solutions.

Monitoring whether service-level agreements (SLAs) conditions are fulfilled by WSs is a problem related to monitoring WSs for changes. In the study by Goel and Shyamasundar (2010), a run-time monitoring framework which allows accessing exchanged messages and comparison against designed scenarios is presented. The focus is on quality of service (QoS) aspects, for example, of a time-out mechanism detecting unavailability of the service. In the study by Goel *et al.* (2011), emphasise is put on detection of violations at the functional level. SLAs are described formally using temporal logic and are used to verify the behaviour of WSs at runtime, for example, maximum response time.

The shortcoming with the above-mentioned solutions lies in the fact that they demand specific knowledge on the kind and the nature of the WS. Also, the kind of change that will be monitored is required to deploy the proper solution. In many scenarios, however, only the URL and interface of the service are known, but no information on whether the WS is conversational, stateful, deterministic, etc.

The WS monitoring framework (WSMF) presented by Miksa *et al.* (2015) is thus designed to allow investigation of any kind of WS, and to facilitate reasoning about the nature of a service. If the WS is deterministic, the monitoring process can be launched and all four types of changes (Table I) can be detected. Otherwise, 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 WS extensions

Apart from monitoring the technical level of services, several improvements to the specification of WSs, which should lead to a higher sustainability of processes, and reduction of the need for continuous monitoring have been proposed.

The Universal Description Discovery and Integration (UDDI) is a registry which holds information on registered WSs. However, the registry does not contain sufficient additional information on the service that would allow the user to obtain information on the nature, behaviour or QoS. Several proposals aim at enriching the purely functional description of WSs (bindings, ports, etc.) with QoS aspects, for example, timing aspects, availability, reputation (Comuzzi and Pernici, 2009) and pricing (Liu *et al.*, 2004). W3C Working Group (2003) specifies requirements for QoS for WSs. It lists 13 points which should be fulfilled, but none of them concerns guaranteeing continuity or non-modifiability.

Another approach is to facilitate versioning of WSs. Yet, in this case, approaches do not aim at specifying a way to interweave versioning into WS specification, but present workarounds to deal with the currently underspecified WS standards (Kaminski *et al.*, 2006). One of the exceptions is the study by Kalali *et al.* (2003), which provides functional requirements for a registry which notifies clients when a version of an interface changes. Kaminski *et al.* (2006) is a good example of the current common view on versioning: versioning is understood as a change of interface. Changes in functionality while the interface stays the same are not considered. In Miksa *et al.* (2015), we thus introduced a concept of RWS, which aims at extending specification of WSs, addressing the challenge of functional changes. They are discussed in detail in Section 4.

2.4 Analysis of changes

For implementation of the on-site monitoring, described in Section 6.2, we use the context model, a model that allows for a structured storage and analysis of the descriptive meta-data and documentation of the system (Mayer *et al.*, 2015, 2014). This model allows to store information of a complete business process, from the sequence of steps executed to the technical infrastructure (software and hardware) that supports it. It defines a set of concepts that can be used to describe the process. The concepts can easily be extended, if a specific use case requires so. The model architecture is depicted in Figure 1. The model allows for reasoning on the information, thus being able to identify which steps in a process depend on an external service, or which specific hardware and software is providing the functionality for another step. This allows for a detailed analysis of which parts of a system are affected by various changes.

3. WS monitoring framework

The WSMF, introduced in Miksa *et al.* (2015), 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 service is intercepted and stored.

(2) *Transform*: Requests and responses are grouped and enriched with additional meta-data.

(3) *Reason*: Data are analysed, and a type of the WS 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 WS 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

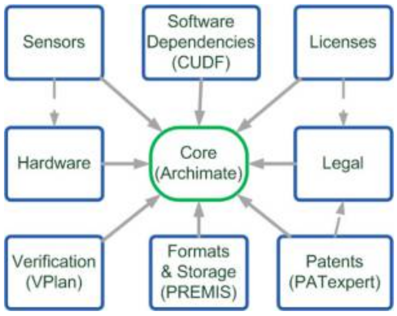


Figure 1.
An overview on the process context model: core ontology and extensions

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 WS 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 WSs are deployed primarily for information transformation, collection or computational services.

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 2 depicts the communication paths between the involved parties in the proxy mode. Figure 3 shows the proof of concept tool that implements the WSMF. 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 RWS as an extension to the WSDL WS specification. We describe methods that are required to transform a regular WS into an RWS.

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.

In Miksa *et al.* (2014), we defined principles of RWS design and elaborated a list of methods that constitute RWS. For the sake of the paper's completeness, we provide below the description of RWS methods and also examples of responses obtained from these methods.

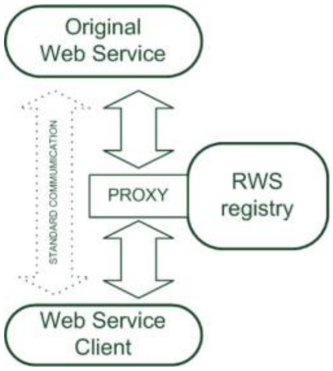
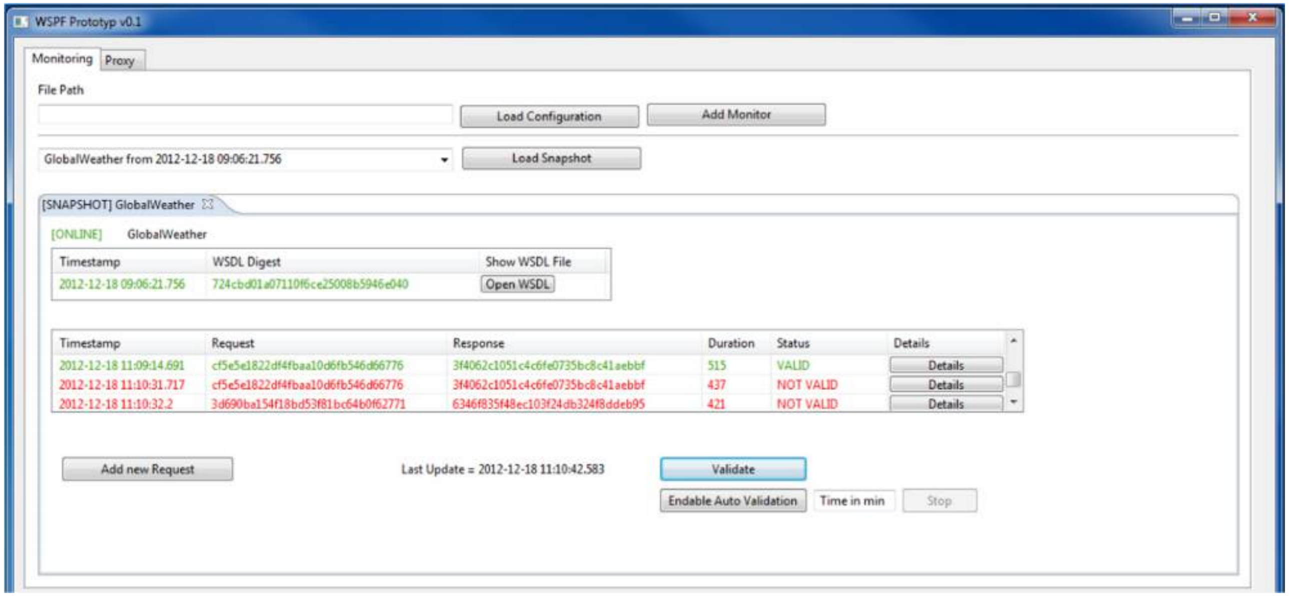


Figure 2. Communication between parties involved using the WSMF framework in a proxy mode



GlobalWeather from 2012-12-18 09:06:21.756

[SNAPSHOT] GlobalWeather

[ONLINE] GlobalWeather

Timestamp	WSDL Digest	Show WSDL File
2012-12-18 09:06:21.756	724cbd01a07110f6ce25008b5946e040	Open WSDL

Timestamp	Request	Response	Duration	Status	Details
2012-12-18 11:09:14.691	cf5e5e1822df4fbaa10d6fb546d66776	3f4062c1051c4c6fe0735bc8c41aebbf	515	VALID	Details
2012-12-18 11:10:31.717	cf5e5e1822df4fbaa10d6fb546d66776	3f4062c1051c4c6fe0735bc8c41aebbf	437	NOT VALID	Details
2012-12-18 11:10:32.2	3d690ba154f18bd53f81bc64b0f62771	6346f835f48ec103f24db324f8ddeb95	421	NOT VALID	Details

Last Update = 2012-12-18 11:10:42.583

Figure 3.
Software support for the WSMF, including a proxy mode

4.1 minAvailabilityDate()
This method has no input parameters, and 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.

4.2 identifyYourself()
This method has no input parameters. The output consists of multiple information as follows:

- *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 and StatefulNonDeterministic.
- *First release data*: Date since when the WS is available;
- *Total number of changes*: Integer indicating how many changes to the WS have been detected since its beginning – it includes both expected (new versions) and unexpected changes.
- *Availability percentage*: Availability of the WS expressed in percentage.

4.3 getSystemEnv()
This method has no input parameters. 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 (PREMIS Editorial Committee, 2008). 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, whereas the concept of software also has 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 concepts, 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 WS owner who is providing the service on serviced infrastructure.

4.4 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 I.
- *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 5 presents an excerpt of the XML Schema Definition (XSD) schema defining the format of the response sent by this method. The types of changes are encoded as

```
<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:enumeration value="NoEffect" />
   </xs:restriction>
</xs:simpleType>
```

Figure 4. Fragments of an XSD schema defining format of the response for *getChangesSince* method

```
<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 5. Description detailing the changes made to the system by replacing *Oracle Java* version 1.6 with 1.7

enumerations, and the ontology with a list of changes is provided as a string directly in the body of the response.

4.5 getContact()

This method has no input parameters. 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 RWS 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 sends notifications to the service consumer. The main task of the registry is the decoration (design pattern) of existing WSs with resilient methods. The registry is a service provided by a third party. Such an approach should substantially increase the acceptance of RWS 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 6 illustrates the process of converting a WS into an RWS using the registry. There are three actors involved: the WS provider, the WS consumer and the registry operator. When using traditional WSs, 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 WS, by verbatim copying information about the WS 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 consist 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 WS. 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 WSMF described in Section 3. However, the implementation described in Miksa *et al.* (2015), which uses network packet capturing 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 the study by Bartolini *et al.* (2009), 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 use of the aforementioned monitoring tool using the proxy mode.

We simulated changes on a number of WSs for which we had access to the source code, to detect all types listed in Table I.

5.2 On-site monitoring

In a second approach towards implementing RWS, we deploy certain components directly on site of the WS 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 WS. 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 (Table II).

Most of the resilient methods introduced in Section 4 do not require an implementation specific to a given WS. As such, the hardware or software setup of the

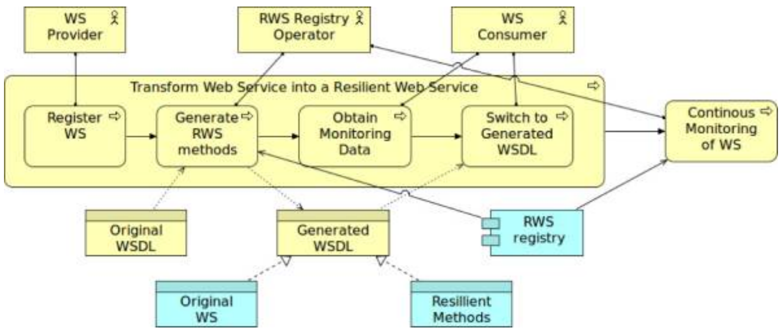


Figure 6.
Business process model, depicting the transformation of a WS into an RWS

machine can be determined via methods that are generic, regardless of the actual implementation of the WS, but can be determined in the same or similar fashion for all WSs using, for example, the same type of operating system. Some methods may depend on the specific programming language used for implementing a WS, 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 WSs 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 an RWS.

In Miksa *et al.* (2014), we provided an example of such a resilient Application Programming Interface (API) for the Java programming language using publicly available frameworks like platform-independent SIGAR framework[2], or utilities such as LHW[3] for Linux. In Section 6.2, we demonstrate another way in which on-site monitoring can be implemented.

5.3 Client side implementation

Once the WS has been upgraded into an RWS, the service consumer needs to implement support for resilient methods. This implementation can, for example, be similar to the exception handling mechanisms used in programming languages.

Because of the fact that the WS 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 WS 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 RWS. An alternative implementation may use push-style notifications to propagate information on changes, for example, email. 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 Simple Object Access Protocol).

Table II.
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

6. Use case-specific implementation

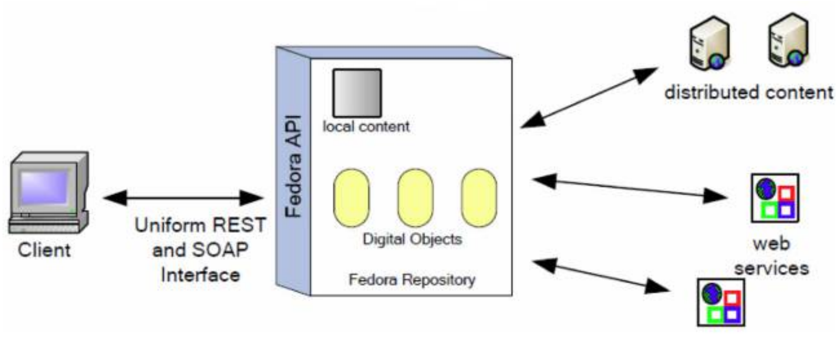
In this section, we describe application of RWS on-site monitoring on a Phaidra repository system. We chose this use case because it is a representative of other repository systems that are also built following SOA. The domain of digital preservation, in which repositories operate, puts special attention to maintaining the high level of trust that the accessed objects are faithfully presented and, therefore, can be interpreted correctly.

In Section 6.1, we describe the architecture, role of WSs and the organisational setting of the use case. In Section 6.2, we describe how we introduced the RWS and the WSMF into the existing system, while minimising the additional effort by reusing as much as possible already existing system components and models.

6.1 Phaidra repository system

The repository system described here is a university-wide digital asset management system with long-term archiving functions. Students, researchers and co-operators with the proper authorisation can upload and link the objects which, among others, can be text, image and audio files in multiple formats.

6.1.1 Architecture. The system consists of two main components, the backend and the frontend. The backend is realised with the use of Fedora Commons[4], which is an open-source system that allows for storing, managing and accessing digital objects. The web frontend is responsible for the presentation of contents or editing of metadata. The frontend was developed at the university. The frontend interacts with the Fedora repository through the Fedora API, as seen in Figure 7. The backend may also interact with other systems to obtain the content stored on different servers (distributed content) or may use WSs to get additional information about the contents or to perform data transformation (e.g. format conversion, video streaming). These services are of particular interest because they may change in different ways and, therefore, alter the information and the content delivered to the end users of the repository system. For example, a scientific paper can be stored in the Fedora repository in three formats – HTML, PDF and TEX – and all of them will be grouped under one digital object. No disseminators (services performing operations on content) will be used to transform the content because the content will be provided at the moment of creation of the digital object. However, the same final



Source: Fedora Commons Community (2007)

Figure 7. Backend (Fedora) as a mediator for services and content

(visible to the end user) result could be obtained with the use of disseminators. It would be possible to store, for example, only the TEX file and use WSs to generate on demand a PDF or HTML version of the document. The second solution introduces dependency on the services (disseminators). Such dependencies are unavoidable in case of interactive contents, like interactive art, games and computer programmes, which need a special environment to render these artefacts, or when the original file format becomes obsolete and must be transformed into the one that user can work with.

One of the services in use is the image converter that is used every time users access a web page with a summary about a digital document. For example, if the user browses through a collection of PDFs and opens one of them, then they are presented with a preview of the first page of the paper which is a PNG file generated from the first page of the original file. This is achieved with the use of ImageMagick[5] and the corresponding Perl module which needs to be installed in the operating system underlying the repository system. If a different version of ImageMagick is used, it may happen that the conversion may result in a different output. Other examples of services are the video streaming service and the book viewer service.

6.1.2 Organisational context. Phaidra can be used by any institution on the condition that this institution joins the community of Phaidra users. The community is a forum to share experiences and work jointly on improving the system. Phaidra is a common product for all of the stakeholders, but each of them runs their own installation. Each of them may have different requirements and, therefore, may develop different features (services) independently, but is always obliged to share these new functionalities with other members. Therefore, it is possible that some of the WSs hosted at one of the instances of the system are used by one of the other partners. The consortium agreement does not specify any formal agreements considering SLA of shared services. To ensure that any changes that may occur in the service hosted by other consortium member do not affect the given Phaidra installation, a mechanism that monitors the service is needed. Therefore, we proposed to use the RWS and WSMF.

6.2 RWS on-site monitoring

In this section, we describe how we implemented the on-site monitoring for an existing WS, thus converting it into an RWS. We present actions not only taken on the side of the service provider but also describe in what way the service consumer can automatically process the information provided by the RWS methods. For this experiment, we used two virtualised instances of Phaidra: one for the service provider and one for the service consumer. Thus, we have two independent instances having one common service, which is image converter service.

6.2.1 Available auxiliary resources. We start by describing already existing resources that will be used during the implementation of RWS mechanisms, namely, an instance of the TIMBUS context model and unit tests.

The structure of Phaidra is represented using the context model. It provides a comprehensive view of the system, taking into account the high-level business layer, which depicts the typical usage scenarios of the system, and also the infrastructure layer, which describes particular software and hardware needed to perform these steps. Thanks to such a binding, one can see which resources are needed to perform particular steps. For example, one can identify when the image converter service is used and what libraries and software are involved to operate the service. The business-level components of the model were modelled once at the beginning when the model was created, the infrastructure-level components are provided by software extractors that analyse the system and encapsulate the information about current configuration into dedicated ontologies that are later linked to the initially created model. The model is updated always when any change takes place, and hence, it corresponds to the current state of the system. Owing to the fact that the context model already existed, we decided to use it for implementation of RWS to minimise the effort.

For the reason that the major part of the system (frontend) was developed by one of the consortium members, we were also provided with unit tests that are used to verify the systems functionality and validate the correctness of returned results. The tests are implemented using JUnit[6], a Java framework for test automation, and using Selenium[7], which is a framework for automation of interaction with the web browser, but also capable of validating website contents, taking screenshots, etc. The tests scenarios overlap to a great extent with the typical usage scenarios that were modelled in the context model business layer. Later in this section, we describe how we use these tests on the consumer side of the RWS.

6.2.2 RWS deployment. We started the transformation of the image converter service into an RWS, by converting it from the REST WS into a WSDL WS. This required the creation of a WSDL file and minor changes in the system setup, so that a different type of WS can be used. At this step, we already included the RWS methods in the service WSDL file. All these changes were reflected in the Phaidra's context model.

In the next step, we deployed the WSMF on the Phaidra server. The WSMF was configured to capture live traffic from the network interface and to collect the evidence for the analysed WS. Thus, we created the ground truth data that are later needed for the evaluation of the impact of changes on the WS performance. The results of this assessment are used by the *getChangesSince* RWS method to provide information about changes. We used the context model of the system to provide information about its environment through the *getSystemEnv* method. Obviously, we did not expose all of the information from the context model because this would not only be very insecure but would also simply overload the consumers with details potentially not important to them. Instead, we used Apache Jena[8] and a set of pre-defined SPARQL[9] queries that filter the information about core components of the system and their versions. Furthermore, only these resources that have impact on the functioning of the service are provided. Thus, information relevant only for the given WS is provided. We use the fact that the context model ontology has mapping to the PREMIS ontology, and therefore, we can easily convert the results of queries into corresponding format.

On the consumer side, we adapted the system to use the WSDL WSs (by copying the related modified components from the provider instance) and added implementation making use of RWS methods of the provider. We introduced systematic polling for changes by calling *getChangesSince* method. When the returned result states that there was a change, but the performance of the RWS is unchanged, then no actions are triggered. This could be the case, when the system is modified, but the change of components has no impact on the RWS. In cases when, the change was detected, the method triggers execution of JUnit test cases that check whether this alteration has impact on the final experience of the user. For example, the change resulted in slower execution of the service, but the threshold defined in the unit tests has not been reached, and therefore, the alteration is acceptable. Finally, in case when there was a change detected and the unit tests fail, then a notification is sent to the consumer instance operator, including the change description and list of changes of the RWS.

6.2.3 Experiment. To test the proposed set up, we performed an experiment by updating all Debian packages of the operating system underlying the image converter RWS. Being logged in as root user, we performed the *aptitude update* command and then *aptitude safe-upgrade*. As a result, we updated 148 packages. We ran Debian packages extractor to identify these changes. There is no need to run other extractors because the other parts of the system, for example, Phaidra's code, itself were not modified. However, in settings when we cannot determine to what extent the environment can differ, all extractors have to be run again. The output of the Debian packages extractor provides an ontology describing all installed packages (including their cross-dependencies) in the system.

To identify which packages that have been modified could have an effect on the RWS, we run a tool for comparison of ontologies. We compared the ontology describing installed packages before and after updating system. The tool produced two output files, first one with individuals added to the original ontology and the second one with the individuals removed from the original ontology. For each individual representing a package that was added to the system, we checked in the updated context model, whether the individual was a dependency to the RWS, if so, then it was added to the list of changes that would be provided through RWS methods if a change was detected. In the next step, the WSMF execution was triggered, the requests collected in the ground truth data were re-sent and the obtained responses were compared with the ground truth data responses. The whole process starting from running the extractor through making an ontology comparison and identifying packages that may affect the RWS was scripted and performed automatically.

During our experiment, we identified a *perlmagick8-6.7.7.10-5+deb7u2* package that could have an impact on the image converter service, but the WSMF did not detect any changes. Therefore, the RWS methods classified this as a change with no effect. On the consumer side, the active polling detected the change, but triggered no actions because the change is not affecting the performance of the system.

The experiment has shown that there are many ways in which the RWS can be implemented. We demonstrated that the reuse of already existing tests and system models can significantly ease and facilitate RWS implementation. In other systems and other applications, potentially different set of auxiliary resources is available that can be used to implement the RWS.

7. Discussion

There are several different usage scenarios for the deployment of RWS, 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, etc.) is missing.

Another distinction might be if the WS 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 WS provider would become a consumer of this information from the hosting provider.

WSs are sometimes used to create mash-ups combining functionality of various WSs, thus providing a new service. If the RWS are used to create such a mash-up, then it is possible to forward information on changes from the RWSs used to create it to the mash-up. Hence, the mash-up is also an RWS. In this scenario, the mash-up owner uses the *getChangesSince()* method to receive notifications on changes from the dependent RWSs. Such an RWS chaining is possible regardless of the implementation of the RWS. The mash-up owner needs to design mechanisms to handle notifications from dependent RWSs and add the resilient methods to the interface of his mash-up. The mash-up 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 WS 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.

8. 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 focussed on WSs as one typical representative. We analysed potential changes stemming from WSs that impact business continuity. Monitoring and testing of WSs, as well as extension mechanisms enriching the WSs with additional information on their behaviour and availability were investigated. Furthermore, a use-case implementation for a digital repository system was presented.

Our work put special attention to the recently proposed extension of WSs, namely the RWS. We provided a detailed specification of the resilient methods, ans described two alternative implementations that should ease its uptake and make the deployment easier. The first approach using an external RWS registry allows converting any WS into an RWS without modification at the service provider site. For that purpose we provided an enhanced implementation of the existing WSMF that uses proxy mode to intercept communication. The second approach is an on-site monitoring tools suite that enables full utilisation of resilient methods including information automatically collected from the underlying system.

We tested this solution by applying it in a use case of a digital repository system, where that repository system uses WSs to perform some if its tasks. We monitored the impact of the system libraries updates on the performance of the WS using the monitoring framework to identify any changes that may affect the consumers using the RWS. By using the Context Model of the system, we were able to provide information on changes relevant to the specific consumers. Both of the solutions are capable of providing notification on changes to the WS consumer and thus contribute significantly to the minimisation of the impact of changes in the ICT infrastructure on the business processes. Although the discussion in this paper focussed on WSs, the solutions proposed here can also be applied in other implementations of distributed computing environments.

Currently, we are working on a catalogue of policies that express requirements for the WS owners who are willing to convert their services into RWS. Thus, we want to ensure that all of the independent implementations of RWS conform to its definition and provide reliable information to its consumers.

Notes

1. http://id.loc.gov/ontologies/premis.html#
2. www.hyperic.com/products/sigar;
3. http://ezix.org/project/wiki/HardwareLiSter
4. www.fedora-commons.org
5. www.imagemagick.org
6. JUnit: http://junit.org
7. SeleniumHQ: www.seleniumhq.org
8. Apache Jena: https://jena.apache.org
9. SPARQL: www.w3.org/TR/rdf-sparql-query/; July 1, 2015.

References

Bartolini, C., Bertolino, A., Marchetti, E. and Polini, A. (2009), "WS-Taxi: a WSDL-based testing tool for web services", *ICST 09 International Conference on Software Testing Verification and Validation*, *Denver, CO*, pp. 326-335.

Cao, T.D., Castanet, R., Felix, P. and Morales, G. (2011), "Testing of web services: tools and experiments", *Services Computing Conference (APSCC), 2011 IEEE Asia-Pacific, Washington, DC*, pp. 78-85.

Cao, T.D., Felix, P., Castanet, R. and Berrada, I. (2010), "Online testing framework for web services", *Third International Conference on Software Testing, Verification and Validation (ICST)*, *Paris*, pp. 363-372.

Comuzzi, M. and Pernici, B. (2009), "A framework for QoS-based web service contracting", *ACM Transactions on the Web*, Vol. 3 No. 3, pp. 1-52, available at: http://doi.acm.org/10.1145/1541822.1541825

Dranidis, D., Ramollari, E. and Kourtesis, D. (2009), "Run-time verification of behavioural conformance for conversational web services", *7th IEEE European Conference on Web Services (ECOWS)*, *Eindhoven*, pp. 139-147.

Fedora Commons Community (2007), "Fedora commons tutorial 2: getting started: creating fedora objects using the content model architecture", Technical Report, available at: http://fedora-commons.org/documentation/3.0b1/userdocs/tutorials/tutorial2.pdf

Goel, N., Kumar, N.N. and Shyamasundar, R. (2011), "SLA monitor: a system for dynamic monitoring of adaptive web services", *9th IEEE European Conference on Web Services (ECOWS)*, *Lugano*, pp. 109-116.

Goel, N. and Shyamasundar, R. (2010), "Automatic monitoring of SLAs of web services", *Services Computing Conference (APSCC), 2010 IEEE Asia-Pacific*, *Tokyo*, pp. 99-106.

Kalali, B., Alencar, P. and Cowan, D. (2003), "A service-oriented monitoring registry", *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, *Toronto, Ontario*, IBM Press, pp. 107-121, available at: http://dl.acm.org/citation.cfm?id=961322.961340

Kaminski, P., Müller, H. and Litoiu, M. (2006), "A design for adaptive web service evolution", *Proceedings of the 2006 International Workshop on Self-Adaptation and Self-Managing Systems, SEAMS '06, ACM, New York, NY*, *Shanghai*, pp. 86-92, available: http://doi.acm.org/10.1145/1137677.1137694

Liu, Y., Ngu, A.H. and Zeng, L.Z. (2004), "QoS computation and policing in dynamic web service selection", *13th International World Wide Web Conference*, *New York, NY*, ACM, New York, NY, pp. 66-73, available at: http://doi.acm.org/10.1145/1013367.1013379

McCoy, D.W. (2002), *Business Activity Monitoring: Calm Before the Storm*, Gartner Research, Vol. LE-15-9727 (1 April 2002).

Mayer, R., Miksa, T. and Rauber, A. (2014), "Ontologies for describing the context of scientific experiment processes", *Proceedings of the 10th International Conference on e-Science*, *Guarujá, SP*.

Mayer, R., Antunes, G., Caetano, A., Bakhshandeh, M., Rauber, A. and Borbinha, J. (2015), "Using ontologies to capture the semantics of a (business) process for digital preservation", *International Journal of Digital Libraries (IJDL)*, Vol. 15, pp. 129-152, available at: www.springer.com/-/7/7e9c68c9a6ac468aaacd08a7827e82bf

Miksa, T., Mayer, R. and Rauber, A. (2015), "Ensuring sustainability of web services dependent processes", *International Journal of Computational Science and Engineering (IJCSE)*, Vol. 10 Nos 1/2, pp. 70-81.

Miksa, T., Mayer, R., Unterberger, M. and Rauber, A. (2014), "Resilient web services for timeless business processes", *Proceedings of the 16th International Conference on Information Integration and Web-based Applications & Services (iiWAS2014)*, *Hanoi*, pp. 243-252.

PREMIS Editorial Committee (2008), "Premis data dictionary for preservation metadata", Technical report, PREMIS Editorial Committee.

W3C Working Group (2003), "QoS for web services: requirements and possible approaches", available at: www.w3c.or.kr/kr-office/TR/2003/ws-qos/ (accessed 30 July 2014).

About the authors

Tomasz Miksa is Researcher at SBA Research. Currently, he is involved in the preservation of business processes in the European Union-funded FP7 project TIMBUS. He is also a student of the Vienna PhD School of Informatics, where he is conducting his research on e-science, digital

preservation and research infrastructures. Tomasz Miksa is the corresponding author and can be contacted at: miksa@ifs.tuwien.ac.at

Rudolf Mayer is Senior Researcher at SBA Research. Previously, he has worked at the Vienna University of Technology, where he has been involved in numerous national and international research projects on information retrieval, machine learning and digital preservation, including DELOS, MUSCLE or PLANETS. His current research focus lies on the preservation of processes in the European Union-funded FP7 projects TIMBUS and APARSEN.

Andreas Rauber is Associate Professor at the Department of Software Technology and Interactive Systems at the Vienna University of Technology. His research interests cover the broad scope of digital libraries and information spaces, including specifically text and music information retrieval and organisation, information visualisation, as well as data analysis, neural computation and digital preservation.