Improving the Dependability of Web Services Integration

Yuhui Chen and Alexander Romanovsky
Newcastle University, UK

The WS-Mediator framework employs an off-the-shelf mediator architecture and resilience-explicit computing in pursuit of dependable, dynamic Web services integration.

Web services\(^1\) and service-oriented architectures (SOAs)\(^2\) represent a new paradigm for building distributed computing applications. Web services offer advantages over conventional distributed computing middleware platforms, such as the Common Object Request Broker Architecture (Corba), that make them a critical technology for developing e-science and e-commerce applications. Web services’ loosely coupled architecture, combined with their standardized interoperability, lead to a new computing paradigm that supports the construction of more flexible and dynamic distributed applications.

Thanks to the benefits they provide, Web services are attracting growing numbers of supporters. As a new technology, however, they bring with them both opportunities and challenges. Despite their potential, Web services face a range of problems. In particular, ensuring dependability has become an active area of research in recent years.

In this article, we propose an approach to improve the dependability of Web services integration using an off-the-shelf mediator architecture\(^3\) to support resilience-explicit\(^4\) dynamic Web services integration. To evaluate this approach, we have implemented a platform-independent Java framework based on the WS-Mediator concept, which can easily be integrated into any implementation of Java Web services applications. The framework comes as an additional layer on top of the Java Web service middleware. Our experiments demonstrate the applicability of the approach for real-world Web service applications.
Web Services Dependability

Web services implement capabilities and functionalities via computer networks, especially the Web. A Web service application typically integrates multiple remote services via the Web. Some of those services are dynamically discovered via UDDI.

This kind of integration raises problems that can potentially undermine the application’s dependability; it could then become untrustworthy because of unreliable individual component services or networks.

Research on the topic typically focuses on ensuring these services’ dependability and the communication between clients and service providers. In a real-world application, however, it’s impossible to guarantee that an individual Web service or network will never become untrustworthy. More importantly, clients’ views on the same individual Web service can differ because of the distributed architecture.

Some solutions take advantage of the loosely coupled architecture of Web services to implement a service-redundancy strategy. Unfortunately, such mechanisms can tolerate failures of only individual component services. Their efficiency can be further improved by adopting resilient-explicit computing.

WS-Mediator Approach

The WS-Mediator system implements resilience-explicit mediators as Web service intermediaries, or submediators, which are subsystems of the whole system. These are functionally identical and globally distributed to construct an overlay architecture, working independently or cooperatively to achieve resilience-explicit dynamic reconfiguration in order to adapt fault-tolerance techniques and use the available service redundancy. The submediator’s monitoring mechanism monitors Web services and generates resilience metadata representing their dependability attributes, such as response time, failure rate, failure types, and so on.

This metadata’s structure can be designated for particular application scenarios; the resilience-explicit decision-making mechanism integrated in each submediator uses it to dynamically select the most dependable Web service according to the client’s preference. This approach can make the way service redundancy (including service diversity) is used considerably more efficient.

WS-Mediator General Architecture

The WS-Mediator is an architectural solution deployed on a distributed infrastructure between a set of clients and Web services that they access (see Figure 1). In Web services, the distinction between a client and a service provider is blurry because a service provider acts as a client when it invokes other services.

The overlay system consists of multiple, functionally identical submediators that are globally distributed at different geographical locations.

In the WS-Mediator system, submediators accept service invocations from either clients or other submediators. Figure 2 (on the next page) illustrates the submediator’s general structure. Each has a database that stores information about Web services and metadata that represents their resilience behavior. Submediators monitor Web services from different geographical locations.

There are two kinds of submediators: local and remote. Local ones are deployed at locations...
where client applications are running; for example, a local submediator deployed on a LAN can provide adequate services for the LAN’s clients, and a lightweight local submediator tailored for individual clients can be deployed on clients’ personal computers. This is why the generated resilience metadata accurately represents local clients’ perspectives.

Remote submediators are deployed on Internet backbones. They should be relatively dependable for clients using the Internet service providers (ISPs) that depend on them. Each remote submediator is monitored by local and other remote submediators. In each submediator, a database stores information about remote submediators, including their location, ISP, and dependability metadata. Any submediator may invoke one or more remote submediators to implement comprehensive strategies aimed at improving the dependability of services integration. Participating submediators are selected on the basis of metadata analysis. A client might also invoke remote submediators directly to adapt the WS-Mediator framework to its application.

**Dynamic Reconfiguration and Service Policies**

Figure 2 illustrates the submediator’s general structure and components. The dynamic-reconfiguration component is its brain, handling service-execution procedures. The fault-tolerance mechanisms implement fault-tolerance techniques, which can vary when adapted to particular application scenarios. These mechanisms are implemented as execution models, which the clients choose.

To utilize the WS-Mediator system, clients need to prepare essential information available from the UDDI about the target Web services. When necessary, a client can supply a set of Web services as candidates for implementing services redundancy. When a client invokes the submediator, three kinds of information are assembled in the SOAP message:

- embedded SOAP messages representing the client’s request to each candidate Web service,
- an execution policy for each candidate Web service, and
- a global-execution policy.

```xml
<?xml version="1.0" encoding="UTF-8"?>
  <wsp:ExactlyOne>
    <wsp:All>
      <bindingMethod>SOAP11HTTP</bindingMethod>
      <invocationMode>Sync</invocationMode>
      <timeout>20000</timeout>
      <autotimeout>max</autotimeout>
      <retryAfterFailure>3</retryAfterFailure>
      <retryInterval>30</retryInterval>
      <multirouting>0</multirouting>
      <monitorThisWS>no</monitorThisWS>
      <searchIdenticalWS>2</searchIdenticalWS>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

The embedded SOAP messages carry the client’s requests to the target Web services. The execution policy associated with each SOAP message indicates how the submediator should deal with the SOAP message. The WS-Mediator execution policy extends the WS-Policy framework, which in effect is an instruction to the WS-Mediator to execute each particular invocation. Here is an example of a policy:
If the execution policy is omitted, the submediator will use a default execution policy.

The global-execution policy also extends the WS-Policy framework and is an instruction to the WS-Mediator about how to execute the client’s requests. This states, for example, which execution mode to use and how to process the results from the target Web services. The global execution policies are different in different execution modes. Here is an example of the global-execution policy implemented in the Java prototype of the WS-Mediator:

```xml
<?xml version="1.0" encoding="UTF-8"?>
    <wsp:ExactlyOne>
        <wsp:All>
            <wsmgp:AlternativeRedundancy execution="true">
                <priority>dependability</priority>
                <dependabilityAcceptance>50</dependabilityAcceptance>
                <performaceAcceptance>300000</performaceAcceptance>
                <timeout>100000</timeout>
            </wsmgp:AlternativeRedundancy>
        </wsp:All>
    </wsp:ExactlyOne>
</wsp:Policy>
```

Resilience-Explicit Computing
Resilience-explicit computing is a critical concept in the WS-Mediator approach. The WS-Mediator monitors Web services at different locations, with their dependability metadata collected and analyzed only locally by submediators. This data includes availability rate, average response time, maximum response time, and so forth. Therefore, generally speaking, a Web service’s dependability metadata is different in different submediators. A submediator can request the metadata of a particular Web service from other submediators. Here is an example of resilience metadata:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<metadata>
    <ws service="http://xml.nig.ac.jp:80/xddbj/Blast">Blast">
        <dependability>71</dependability>
        <performace>24141</performace>
        <numOfTests>405</numOfTests>
        <succTests>290</succTests>
        <aveResponseTime>24141</aveResponseTime>
        <minimumResponseTime>1110</minimumResponseTime>
        <maximumResponseTime>2750</maximumResponseTime>
    </ws>
</metadata>
```

The resilience metadata generated by the monitoring mechanism is used for resilience-explicit dynamic reconfiguration. For example, in the recovery-blocks execution mode, the Web service with the best metadata is used as the main Web service. The redundant candidates are also sorted according to their metadata.

WS-Mediator Framework
We can use the WS-Mediator architectural solution in different forms to suit specific application scenarios. We implemented a Java prototype of the WS-Mediator to validate the approach’s applicability. The framework consists of the remote submediator and Mediator-Elite components. The former is a standard implementation of the submediator component, whereas the latter is a lightweight local submediator that we implemented as an additional layer between the client application and Web services middleware that individual clients can tailor for deployment.

Our implementation is based on the Sun Microsystems Glassfish platform. Figure 3 illustrates the Java WS-Mediator framework.

The Mediator-Elite is implemented as an off-the-shelf Java package. It provides easy-to-use APIs to help clients utilize the WS-Mediator framework. They can invoke those APIs instead of the existing Java SOAP Web service invocation APIs provided by the Web services middleware packages, such as AXIS, Glassfish, and Soaplab. The WS-Mediator framework greatly simplifies the implementation of invocations of the Web services.

The following example demonstrates how to integrate APIs in a Java Web service client application implementation. As the code illustrates,
the `TestCase()` class implements a simple client application, and method `ws1()` assembles an invocation of Web service “http://xml.nig.ac.jp:80/xddbj/Blast.”

```java
import com.mediator.mediator_Elite.Med_Elite_SOAPPort;
import com.mediator.mediator_Elite.SOAP_Proc;
public class TestCase {
    private Med_Elite_SOAPPort mesp;
    private SOAP_Proc soapProc = new SOAP_Proc();
    ...
    public static void main(String[] args) {
        mesp = newMed_Elite_SOAPPort();
        ws1();
        ws2();
        globalPolicy=readFileCreateDocument("C:\globalPolicy.xml");
        mesp.
        setGlobalPolicy(globalPolicy);
        Vector results =
        mesp.execute();
    }
    private void ws1(){
        QName serviceQName =
        new QName("http://xml.nig.ac.jp:80/xddbj/Blast", "Blast");
        QName portQName =
        new QName("http://tempuri.org/Blast", "Blast");
        SOAPMessage soapMessage =
        soapProc.bindingSOAP(
            (String) smRequest);
        xmlPolicy =
        readFileCreateDocument(
            "C:\ws1_Policy.xml");
        HashMap faults =
        new HashMap();
        faults.put("Result", "busy");
        mesp.insert (serviceQName, portQName, soapMessage,
        xmlPolicy, faults);
    }
    private void ws2(){
        ...
    }
}
```

Figure 3. The Java WS-Mediator framework can be deployed as a remote submediator in a layer between the client application and Web services middleware, whereas individual clients can deploy lightweight local submediators.

(See www.students.ncl.ac.uk/yuhui.chen/ for a more detailed explanation and additional examples.)

**Approach Evaluation with Blast Web Services**

In an earlier work, we presented an experimental study analyzing the dependability of two Blast Web services used in bioinformatics. Blast is an algorithm commonly used in silico experiments in bioinformatics to search for gene and protein sequences that are similar to a given input query sequence. We discovered that the dependability characteristics of Blast Web services were dramatically different. They also varied when monitored at different geographical locations. Our analysis shows that the existing Blast services are likely to offer a reasonable degree of diversity in spite of the fact that they all execute the same basic matching algorithms. This is due to differences in the databases, the specific searches they execute, and the software code they run.

In our latest experiments, we deployed a Java WS-Mediator framework on three Blast Web services. The Blast Web services were hosted by

- the European Bioinformatics Institute (EBI), Cambridge, UK,
- the DNA Databank, Japan (DDBJ),
- the Virginia Bioinformatics Institution (VBI; see http://pathport.vbi.vt.edu/main/home.php).

In these experiments, we implement a Java client application based on the WS-Mediator framework APIs. This application uses the three Web services as candidates to search their Basic Local Alignment Search Tool (Blastn) database for an identical query sequence. This is a database that compares
a given nucleotide sequence to all other nucleotide sequences in the database and then identifies sequences that share regions of homology with the given sequence. Our Java client application sends a request every 30 minutes; if it receives an erroneous reply from a Web service, it retries three times, at 30-second intervals, before switching to a redundant service. The WS-Mediator automatically sets the three services’ timeout periods according to their maximum response time recorded in its database.

We used recovery block, N-version programming, and multirouting execution modes in the experiments and logged the XML reports for analysis.

Figure 4 gives the results we obtained in the recovery-block mode. At the beginning of this run, the WS-Mediator dynamically ordered the three Blast Web services by their dependability rates during the preceding execution according to the global-execution policy. The DDBJ was used as the primary Blast Web service because it was the most dependable. At some points during the execution, however, when the DDBJ reported that the search and analysis service was busy, after trying again three times, the WS-Mediator switched to using VBI. After that, the VBI returned valid results in most attempts. Because the DDBJ was not in a dependable state, its dependability rate dropped dramatically. Starting from point A in Figure 4, the VBI became the most dependable Web service and was therefore chosen as the primary Web service.

There is an interesting contrast between the two switching sequences during the invocations. As Figure 4 shows, there were two entirely failed executions during the experiment. In the first one (point B), the DDBJ was the primary Web service called, the VBI the second, and the EBI the last. In the second execution (see point C), the VBI became the primary Web service called, followed by the DDBJ. The EBI was still the last service attempted. The logged metadata that the monitoring mechanism generated ensured that the switching sequences were correct according to the dependability metadata at the time. In this execution mode, the average overhead of the Java WS-Mediator framework was only approximately 100 milliseconds (ms).

Figure 5 shows a proportion of the results collected in the N-version programming execution mode. In this experiment, all three Web services were invoked simultaneously. Once a Web service obtained the quickest result, the execution terminated. This strategy is slightly different from the classic N-version programming technique, which commonly requires voting on results. However, in a real-world
application, it is not always possible to vote on results received from different services. Therefore, in the Java WS-Mediator framework, voting is an implementation option, but it is not defined by the global policy.

As Figure 5 illustrates, because the DDBJ and the EBI were, for unknown reasons, in unstable states, they failed to provide valid results in response to the invocation. The final results of all executions were returned from the VBI. In this execution mode, the overhead of the Java WS-Mediator Framework is approximately 130 ms—slightly higher than that in the recovery-blocks mode.

Although the evaluation experiments have demonstrated the potential of our WS-Mediator system, we continue to work on improving the system’s functionality and applicability, including the enhancement of monitoring and explicit-computing algorithms as well as enriching fault-tolerance mechanisms. At the same time, we are modeling a detailed overall architecture of the WS-Mediator system and its components. We envision deploying a general WS-Mediator system for public access and evaluation as well as integrating the WS-Mediator system into a real-world e-Science application.

More information about our research and the downloadable WS-Mediator Java package are available at www.students.ncl.ac.uk/yuhui.chen/.

Acknowledgments
Alexander Romanovsky is partially supported by the FP6 ICT Rigorous Open Development Environment for Complex Systems (RODIN) and EPSRC/UK Trustworthy Ambient Systems (TrAmS) projects.

References
7. SOAP, version 1.2, W3C, recommendation, Apr. 2007; www.w3.org/TR/soap/.

Yuhui Chen is a PhD student at Newcastle University, UK. Contact him at Yuhui.Chen@ncl.ac.uk.

Alexander Romanovsky is a professor in the School of Computing Science at Newcastle University, UK. Contact him at alexander.romanovsky@ncl.ac.uk.