FOUNDATIONS OF COMPUTING AND DECISION SCIENCES
Vol. 36 (2011) No. 3 – 4

# INTEGRATION OF HETEROGENEOUS WEB SERVICES IN EXCEPTIONAL SITUATIONS

Paweł L. KACZMAREK*

**Abstract.** Web services are intended to enable interoperability between heterogeneous distributed systems. Although the technology has been widely adopted and accepted, there are still differences between runtime platforms in exception structure and handling. This results in difficulties in effective handling of exceptions during Web services invocation. The paper presents a solution that enables coordinated exception handling between different environments, which involves communication between the client and server to exchange exceptional information and invocation of defined handling functions. The functions are supplied by dedicated libraries that extend heterogeneous runtime platforms. Additionally, IDE environments are augmented with facilities for development of Web services exception processing code. An implementation of the solution for IBM WebSphere Application Server and Microsoft Internet Information Server is presented

**Keywords:** exception handling, service integration, heterogeneous environments, self healing, Web services

## 1. Introduction

The open environment of Service-Oriented Architecture (SOA) encourages the use of existing components for application development. Integration of components-off-the-shelf enables reducing development cost and time by using already existing modules. The components are often run in heterogeneous environments, which makes interoperability issues especially important. Web services standards [11] (WS) are used to enable interoperability between components for both similar runtime platforms (Oracle AppServer and IBM WebSphere), and for different runtime platforms (.NET and Java based).

Using an existing component, with a possibly unknown supplier, requires resolution of dependability issues[3]. Many techniques for dependability assurance of SOA systems have been designed such as component rating, failure modeling or fault tolerance [7]. Exception handling is one of the fault tolerance techniques that is used in SOA systems. Although WS

* Faculty of Electronics, Telecommunications and Informatics, Gdańsk University of Technology, Narutowicza 11/12, 80-233 Gdańsk, Poland, pkacz@eti.pg.gda.pl

standards anticipate the necessity of exception notification during service invocation[11][8] and supply interoperability, difficulties in exception handling still exist:

- Exception handling rules differ between runtime environments.
- The reason for exception is often unknown on the client-side. Runtime libraries typically throw few standard exceptions (for example SoapFaultException and RemoteException) that do not immediately indicate the source of the problem.
- Coordinated handling of exceptions is often difficult or impossible.

The paper describes the REHandler (Remote Exception Handler) middleware that supplies mechanisms for automated or coordinated exception handling. REHandler extends runtime platforms with a dedicated service that supports integration of heterogeneous environments. Additionally, existing IDE environments are extended with additional mechanisms for development of exception handling code - IDE Extensions for Exceptions (IDExx). In more detail, the contributions are as follows.

- We present the REHandler middleware and rules for handling of exceptions during invocations. The middleware supplies client-side and server-side modules, which enables both client-only and client-server exception processing.
- We define a method for classifying exceptional conditions in WS communication. We propose three general purpose exception classes SoapNetworkException, SoapServerException, SoapClientException.
- We adjusted and implemented the REHandler for two leading environments: Microsoft Internet Information Server and IBM WebSphere Application Server.

The rest of the paper is organized as follows. The next section overviews REHandler architecture and presents general system operation. Sect. 3 presents information about automated handling. Sect. 4 describes the implementation of the solution. Sect. 5 describes related work. Finally, Sect. 6 concludes the paper and presents directions of future work.

## 2. Architecture and behavior of the handling system

The proposed solution addresses by supplying two complementing mechanisms: extensions for WS runtimes and extensions for IDE environments as shown in Figure 1.

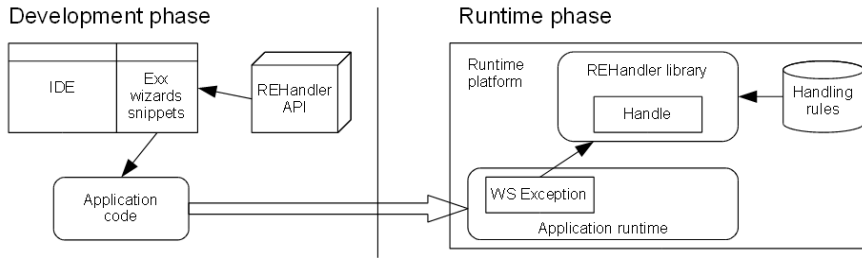


**Figure 1. Integration of REHandler during development and runtime**

The REHandler middleware consists of two complementary modules: a client-side library and a server-side library that are potentially deployed in different environments. If a client service receives an exception, it may request from the client-side REHandler to

initialize exception handling. Server-side REHandler supplies automated exception handling functions that concern either server-only or client-server cooperation.

Processing of exceptions by REHandler is done in the following steps:

- A client-side REHandler retrieves information about the source exception from the SOAP fault (exception) that was sent.
- REHandler decides whether to initialize automated exception handling or to throw the exception to the client. The decision depends on availability of handling functions, exception type and custom configuration.
- Automated client-only handling is selected for exceptions independent from server (for example broken link). The client-side REHandler invokes one of local handling functions.
- Coordinated client-server handling is selected for exceptions from a server error:
    - The client-side REHandler communicates with the server-side REHandler informing about the occurred exception and requesting a repair action.
    - The server-side REHandler performs a repair action in its runtime platform.
    - The client-side REHandler waits for finishing the repair action and repeats (one or more times) the original invocation
- Exception is signaled to the client if it can not be handled automatically.

As an example, suppose a client invokes a Web service that returns an exception. The client-side REHandler processes the exception and identifies SQLException as the source of the exception. Then, the client-side module requests from the server-side module performing a handling action. The server-side REHandler restarts the database. After the restart, the client repeats the invocation of the Web service.

REHandler supplies two kinds of access methods for exception handling: a local interface and a remote interface as shown in Figure 2.

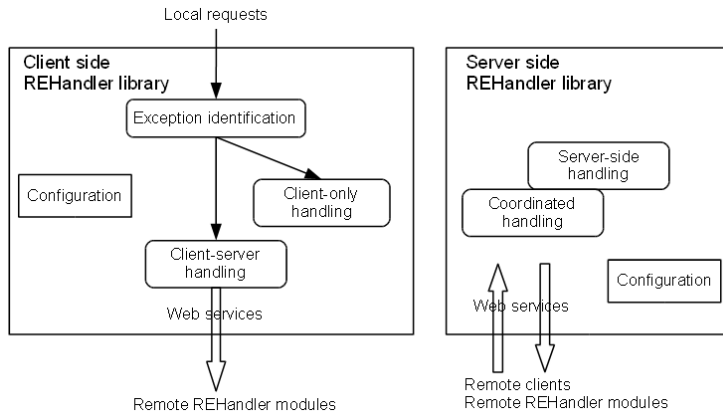


**Figure 2. Main elements of REHandler architecture**

The local interface is used by a local component to request REHandler to process a received exception. The remote interface is used by a remote REHandler module that

requests handling of an exception thrown by a service hosted in the runtime platform of the REHandler module.

REHandler identifies the type of a thrown exception using attributes of a SOAP exception that in turn correspond to sub-elements of the *fault* element of SOAP.

IDE environments are extended with tooling that simplify development of exception handling code in WS invocation - IDExx. IDExx supplies wizards and code snippets that simplify development of handling methods invocation.

## 3. Methods of exception processing and handling

Web services anticipate exception transmission in the SOAP *env:Fault* element and its sub-elements: *faultcode, faultstring, faultactor* and *detail*. *faultcode* indicates whether the Client or Server is responsible for exception occurrence. *faultstring* contains textual information describing the thrown exception, usually a simplified version of stack trace. The contents of *faultstring* is not standardized, and differs between runtime platforms. *detail* may be used to transport detailed and structured information about a custom exception thrown by a WS. A WSDL definition may contain the *fault* element that enables defining which custom data will be transferred if an anomaly occurs.

### 3.1. Identification and classification of exceptions

Typically, libraries for WS invocation throw one of few standard exceptions if SOAP *env:Fault* is received ($exc_{ws}$), for example, SoapException or RemoteException. Attributes or the exception are filled with values from *env:Fault* sub-elements. Client-side REHandler uses attributes of the thrown exception to identify the type of exception that occurred on the server-side (the source exception $exc_{source}$). Parsing of the *faultstring* element for exception names is usually most efficient for identifying both the final and base exceptions. The *detail* element is checked for custom exceptions defined in WSDL:*fault*. Finally, *faultcode* and *faultactor* are read to check the high-level source of the exception, either client or server.

Differences and inconsistencies exist in exception processing during WS invocations. Entirely different environments (such as .NET and JEE) supply different exception programming models and exception classes that are not interchangeable. The work [4] shows experimental studies of WS invocations for two Java-based environments: Sun Microsystems WS Toolkit and IBM WS Toolkit. Significant inconsistencies in exception handling were identified and described. For example, thrown exceptions for the same fault are different in different runtime environments.

Considering the inconsistencies, we propose that Web services invocation libraries throw the following standard exceptions: SoapClientException, SoapNetworkException and SoapServerException that inherits from SoapNetworkException.

- SoapClientException - the exception results from ill-behavior of a client.
- SoapNetworkException - the exception results from a network fault or inability to communicate with the server, for example communication time-out.

- SoapServerException - the exception is thrown if it is known that a SOAP message has been received from the server and a server fault occurred. The returned message contains *faultcode* element set to the *Server*.

SoapServerException inherits from SoapNetworkException as there exist cases in which it is impossible to determine whether the reason for a failure is server or network (for example, a server crashes after receiving a request).

## 3.2. Automated handling of exceptions

The following cases may be distinguished considering client-server cooperation:

- The client-side has not received any information from the server side. $exc_{ws}$ contains only information specific for the local environment.
- The client-side has received some information from the server side. $exc_{ws}$ may contain information specific for the server-side environment, in particular $exc_{source}$

Coordinated handling may be necessary in the first case, as the server side may have received some information from the client. For example server crashed after receiving a request from the client. The second case requires coordinated handling or propagation of the exception depending on exception type.

Automated exception handling in REHandler is initialized by the client-side REHandler middleware after identification of the source exception ($exc_{source}$). Typically, REHandler does not attempt to handle anticipated exceptions that result from client-side faults, with *faultcode* set to *client* and exceptions defined in the WSDL *fault* element.

The middleware enables definition of handling functions and supplies exemplary ones. Logically, three types of handling are distinguished: (i) client-side, (ii) client-server and (iii) coordinated handling.

Client-side handling concerns the client-side REHandler only. This kind of handling is applied if the client is the reason of the exception or the $exc_{source}$ is empty. For example, a client is configured to use an alternative service if it can not locate the original service.

Client-server handling is initialized by a client-side REHandler that requests a repair action from the server-side. For example, the server-side REHandler handles a FileNotFoundException exception by creating an appropriate file in the server file system. Then, the client is notified about the repair action and it repeats the original invocation.

Coordinated handling is initialized if it is required that more parties participate in the process, for example many clients or many servers. REHandler modules are available through WS under defined endpoints, which enables their communication. Handling of SQLException is an example that may require coordination of different parties. REHandler gathers information about thrown exceptions from different clients. If the frequency of exceptions reaches a limit, the database is restarted or reconfigured to use a replica.

Figure 3 shows the communication between runtime environments with both client-side and server-side REHandler modules.

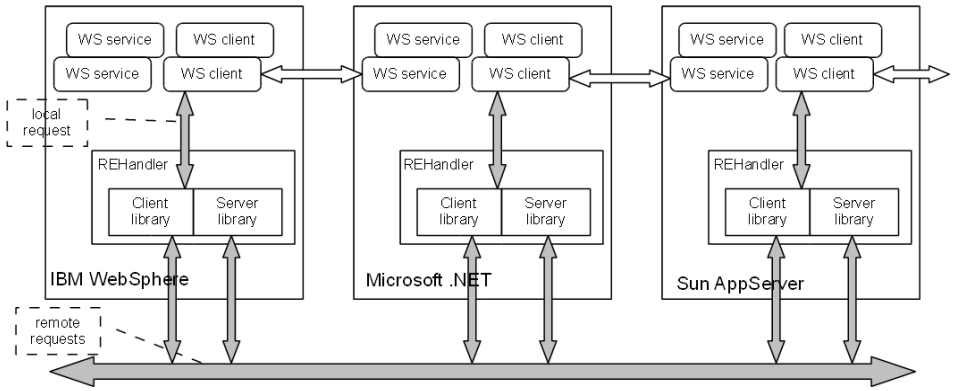


**Figure 3. Communication between Web service modules and REHandler modules**

### 3.3. Mechanisms supporting exception handling

REHandler processes the following types of handling functions:

- Language classes. A handler class implements a common interface with the *handle* method that performs actual handling.
- Operating system scripts. System scripts usually address repairing system-wide issues.
- Reinvocation mechanism. REHandler supplies a built-in mechanism in two variants: a simple reinvocation, and a reinvocation with a delay.

Types of potential exception handling on the client-side include: reinvocation, reinvocation with a delay, N-version invocation of a service, replicated invocation, or selection of alternative services.

## 4. Prototypical implementation of the handling system

The designed solution has been implemented in two environments: Microsoft IIS/.NET Framework 3.5 and IBM WebSphere Application Server 6.5. The implementation is a framework that covers functionality of the designed solution: exception retrieval from SOAP and invocation of handling functions. Exemplary handling functions were included to verify correctness of the solution. Source and binary packages of the current implementation can be found at author web page.

http://www.eti.pg.gda.pl/~pkacz/rehandler.html

Both solutions supply a similar application programming interface and configuration options. We used the Threx prefix in custom names of classes and packages during implementation. Two most important methods include (i) requestRepairAction - handles exceptions from remote requests and (ii) ProcessException - retrieves a specific exception from a Soap exception and performs a handling function. Figure 4 shows the general flow of exception processing and handling in both modules.

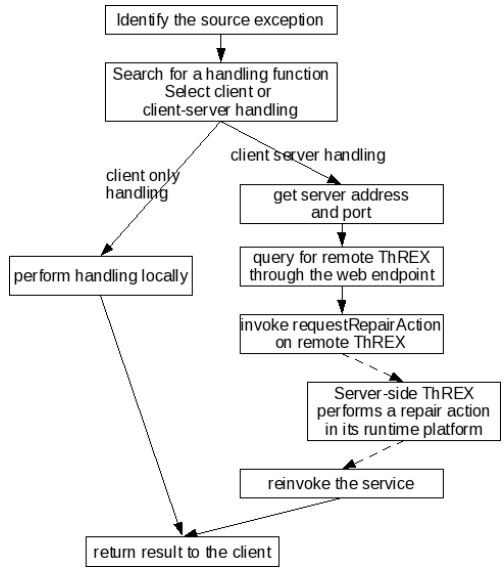


**Figure 4. General flow of exception processing in the REHandler system**

Each REHandler is configured using an XML configuration file. Information about local processing of exceptions concerns: propagation strategy and local exception handling functions (scripts, classes, reinvocation strategies).

The current implementation imposes some limitations to the designed model. The main limitation in J2EE is that some UI elements that have been substituted with code snippets available for application developer. The main limitation of the solution in .NET concerns the support for custom exception types which results from insufficient processing of WSDL files during exception generation.

## 4.1. Exception handler in the J2EE environment

The implementation consists of two main modules: ThrexUtil and ThrexWeb. The first module supplies core functions that enable exception processing on the local runtime platform. The ThrexWeb module supplies the WS interface. ThrexWeb calls ThrexUtil functions to invoke exception handling functions. The J2EE version is implemented using IBM Rational Application Developer and IBM WebSphere AS 6.5.

A configuration file RemoteExceptionsRuleset.xml defines rules for exception processing for both local and remote requests. Currently, the system implements the following types of handling functions: custom code invocation, script execution, request retry, request retry with timeout, generic mapping and remote requests for repair actions.

Client-side application developer uses the HandlerProxy class to invoke remote methods with REHandler support. The solution intends to minimize coding effort as it enables encapsulation of REHandler operations and frees the developer from direct invocation of REHandler methods. Processing depends however on detailed description of individual exceptions in the configuration file, rather than on general configuration.

### 4.2. Exception handler in the .NET environment

The .NET version of the implemented system distinguishes a module for local exception handling (ThrexModule) and a module for WS access (ThrexWS). Additionally, a UI module for Visual Studio supplies code snippets and dialog boxes for inserting Threx-based invocations. The implementation is addressed for the Microsoft .NET framework 3.5, but it may also be applied in the 2.0 version with minor modifications.

A configuration file (ThrexConfig.xml) specifies general settings (propagation strategy, local ports), exception mapping, and repair actions for locally and remotely signaled exceptions. Below we show a snippet of XML-based configuration file for client-side processing of SQLException after receiving it from a remote Web service invocation:

```
<LocalException name="java.sql.SQLException">
  <ProcessingStrategy>
    <RequestRemoteRepair />
    <Wait timeout="100" />
    <Reinvoke />
  </ProcessingStrategy>
</LocalException>
```

The IDExx implementation extends the Visual Studio environment with a new menu option enabling insertion of REHandler exception handling and supplies code snippets for WS invocation. The snippets may be used manually by a developer or configured in the IDExx module to insert the code automatically.

The current implementation has a limited functionality compared to the designed solution. The UI part of the system is not fully integrated with the REHandler part, which results in the necessity of manual code development in some cases.

## 5. Related work

Exception handling in parallel and distributed systems has been researched for years, enabling design of both algorithmic and technological solutions. Programming guidelines for correct exception use in WS have been designed for different programming environments [10]. The guidelines rely mainly on displaying error messages from the *env:Fault* sub-elements as few types of exceptions are available on the client-side. The work [8] proposes a library for processing contents of SOAP *env:Fault* on the client-side. The library is addressed for the Apache SOAP runtime and assumes that stack trace is available in the detail element. Our solution differs in that we address heterogeneous environments and supply distributed handling of exceptions.

The work [12] presents an algorithm that resolves concurrent exception occurrences in distributed systems. The presented algorithm enables recovery of a distributed system if components throw simultaneously different exceptions. The work [5] also focuses on concurrent exception handling using an extended termination. A global exception is introduced that is used to exceptional termination of cooperating processes.

Solutions that enable automated handling of exceptions are usually based on a middleware layer that supplies exception handling or processing functions. The work [9] presents a container-managed exception handling framework that is based on intercepting a

component method call and dispatching exception events at a variety of points during the invocation. High level design of exception handling has been addressed in [2]. The work focuses on UML modeling of exceptions and automated generation of exception management features. The WS-Mediator [1] system addresses a wider problem of dependable service composition. The system employs dependability monitoring and resilience-explicit dynamic reconfiguration of service composition. Our solution differs in that we enhance programming capabilities with exception handlers for WS environments.

Coordinated exception handling in EJB invocations in the J2EE environment is addressed by the author in [6]. The concept of Remote Exception Handler (REH) is proposed that enables invocation of predefined actions in case of exception occurrence. Current work is focused on WS, which results in different exceptional situations and the necessity to handle heterogeneity.

## 6. Conclusions

The presented solution enables automated handling of exceptions during Web services invocation. The solution extends existing runtime platforms with advanced exception processing that should result in an increase of application fault tolerance. The implementation work that was performed in Microsoft .NET and IBM WebSphere gave promising results. The implementation supplies a middleware that enables identification of a source exception, handling of selected exceptions from both local and remote requests, and propagation of exceptions if necessary. The middleware may be extended with detailed exception handling procedures depending on application functionality.

The current implementation may be further extended, which will be the main scope of future work. Dynamic propagation of information about exception handling actions is highly recommended in the system. Now, a REHandler module allows dynamic configuration of its local behavior, but the information is not exchanged with remote modules. Additionally, the .NET part of the system should be extended with processing of WSDL definitions and generation of custom exception types. The Java part should be extended with graphical UI elements that simplify development of REHandler invocations.

### 6.1. Acknowledgment

The author would like to thank students: Łukasz Błoński, Mateusz Bronk and Marcin Sasinowski for implementation work and important remarks concerning practical application of the solution. This work was supported in part by the Polish Ministry of Science and Higher Education under research project N N519 172337.

## References

[1] Y. Chen, *WS-Mediator for Improving Dependability of Service Composition*. PhD thesis, Newcastle University, Newcastle upon Tyne, UK, 2008.
[2] S. Entwisle, H. Schmidt, I. Peake, and E. Kendall, A Model Driven Exception Management Framework for Developing Reliable Software Systems. In *10th IEEE Intern. Enterprise Distributed Object Computing Conference*, 2006.
[3] A. Gorbenko, V. Kharchenko, P. Popov, A. Romanovsky, and A. Boyarchuk, Development of Dependable Web Services out of Undependable Web Components. Technical Report CS-TR-863, University of Newcastle upon Tyne, 2004.
[4] A. Gorbenko, A. Mikhaylichenko, V. Kharchenko, and E.K. Iraj, Exception Analysis In Service-Oriented Architecture. *In 6th Intern. Conf. Information Systems Technology and its Applications, ISTA*, Ukraine, 2007.
[5] V. Issarny, Concurrent Exception Handling. Advances in Exception Handling Techniques, *LNCS 2022*, 2001.
[6] P.L. Kaczmarek, B. Krefft, and H. Krawczyk, Coordinated Exception Handling in J2EE Applications. *In 6th Intern. Conf. on Computational Science, LNCS 3991*, 2006.
[7] ReSIST: Resilience for Survivability in IST, A European Network of Excellence. Resilience-Building Technologies: State of Knowledge, 2006.
[8] R. Shen and E. Choi, Build error-proof Web services. IBM developerWorks, 2002.
[9] K. Simons and J. Stafford, CMEH: Container Managed Exception Handling for Increased Assembly Robustness. In *Component-Based Software Engineering, LNCS 3054*. 2004.
[10] P. Wang and R. Butek, Throw the Right Exception from the Service Endpoint. IBM DeveloperWorks, 2004.
[11] The World Wide Web Consortium, *Web Services Activity*, http://www.w3.org/2002/ws/, 2009.
[12] J. Xu, A. Romanovsky, and B. Randell, Concurrent Exception Handling and Resolution in Distributed Object Systems. *IEEE Transactions on Parallel and Distributed Systems*, 11(10), 2000.