Fault Handling in the Web Service Stack

Oliver Kopp, Frank Leymann, and Daniel Wutke

Institute of Architecture of Application Systems, University of Stuttgart, Germany
Universitätsstraße 38, 70569 Stuttgart, Germany
`lastname@iaas.uni-stuttgart.de`

Abstract. The Web services platform architecture consists of different layers for exchanging messages. There may be faults happening at each layer during the message exchange. First, the paper presents current standards employed in the different layers and shows their interrelation. Thereby, the focus is on the fault handling strategies. Second, current service middleware is reviewed whether and how it follows the fault handling strategies.

1 Introduction

The service-oriented architecture (SOA) is an architectural style for building (enterprise) applications whose building blocks are services. One incarnation of the technology stack required to build SOA applications are Web services [1]. Web services are defined by a modular and composeable stack of standards ranging from low-level communication protocols, over standardized formats for description of services and the messages exchanged during a Web service interaction to high-level standards for defining potentially complex composite applications built from Web services. An important aspect of said enterprise applications is robustness, i. e. applications must be able to cope with faults occurring during run-time.

Although the issue of building robust applications has been addressed in numerous publications (see Sect. 2 for an overview), these typically focus on one specific aspect of fault handling. None of the work has regarded the different layers of the Web service stack altogether. To understand the cause of a fault in the application, it is necessary to understand how the lower levels work and when and how a fault in the lower levels is propagated to the upper levels. Thus, this paper aims at providing an overview of fault handling across all layers of the Web service stack used by an application with special focus on the interplay of the different functions involved in the fault handling process.

To achieve this goal, the contribution of this paper is two-fold: First, we provide an overview of the functionality required to build a service-oriented application and how it maps to different layers in the application's architecture. With this description, we then identify the different fault types that may occur during run-time of such an application and classify them according to the layer they occur on. As part of this description, we discuss different approaches to reacting to a fault both on the level of the employed middleware (i. e. the Web service run-time implementation or the workflow management system) and on the level of the composite application's logic. Second, we provide an overview of how fault handling has been implemented in one open-source technology stack comprising the BPEL [2] orchestration engine *Apache ODE*, the Web service runtime *Apache Axis 2* and the WS-Reliable Messaging implementation *Apache Sandesha 2* and investigate how their implementation relates to identifies fault types.

The structure of the paper is as follows: First, we present an overview of existing work on fault handling in Web service-based applications Sect. 2. An identification and classification of different fault types according to the Web service platform layer they occur on is provided in Sect. 3. The properties of each fault class are discussed in detail on a conceptual level, relating them to existing Web service specifications where appropriate. During this discussion, special focus is placed on pointing out inter-dependencies among faults on different layers. Section 4 complements the conceptual fault classification presented in Sect. 3 by providing an analysis of the fault handling behavior of a workflow management system and corresponding Web service runtime implementation across all layers of the Web service technology stack. Finally, Sect. 5 concludes and provides and outlook on future work.

2 Related Work

Current work on investigating the parts of or the entire Web services platform architecture such as [1, 3, 4] regards the layers in isolation and does not provide an overview on the interplay between these layers.

The Web Service Business Process Execution Language (BPEL [2]) is the de-facto orchestration language for services. It provides concepts for fault and compensation handling. The specification does not state how faults from lower levels of the stack are propagated into the process.

There are several approaches enhancing BPEL engines by adding capabilities of the fault handling. For instance, Jijia et al. [5] present an extension to the invocation handler of the BPEL engine. It can be configured what action is taken in case a Web service fails. Current actions are retry, substitute, ignore and terminate. The authors rely on the infrastructure to propagate network faults to the extension. Modafferi et al. [6] propose enhancements to the architecture of BPEL engines with a similar functionality. Guidi et al. [7] regard synchronous invoke activities: they propose to wait for the reply message regardless of faults in parallel branches in the process before executing the termination handler. Ardissono et al. [8] shows how hypothesis about the cause of a fault can be constructed and how this information can be used in business processes. Friedrich at al. [9] follow a similar approach based on the WS-DIAMOND infrastructure [10]. A summary of all related work in the context of fault handling in the case of Web services is also presented in [9]. All these approaches do not regard the different layers of the WS stack, whereas our work focuses on the interplay between these layers.

The work by Russell et al. [11] presents workflow exception patterns. These patterns investigate the expressiveness of the workflow language and does not deal with the interplay between the workflow layer and the layers below.

To verify the conformance of a BPEL process, the process is represented as a formal system and then verified for properties given by a specification. Current formalizations of BPEL do not take the Web services stack into account [12]. State of the art formalizations such as the Petri net formalization [13] assume asynchronous communication, but do not regard faults in the layers below the interface layer. Lohmann [12] considers lost messages and buffer overflows, but disregards the interplay of the layers in the Web service stack.

A classification of faults with respect to workflows is given in [14, 15]. Here, workflow engine failures, activity failures (expected exceptions), communication failures and unexpected exceptions are distinguished. Workflow engine failures denote failures of the workflow engine itself. Activity failures are also called expected exceptions. They denote that an activity did not complete successfully and hence a special handling is needed. Communication failures are failures in the communication with the activity implementation. This is the focus of this paper. Unexpected exceptions are exceptions on the process definition level, where the structure of the modeled process cannot handle a special case. Mourao et al. [16] show how unexpected exceptions can be supported by special workflows involving humans.

A general taxonomy in the context of dependability is given by Avižienis et al. [17]. Faults in system components cause error states in the system, which manifest in failures. To be in line with the Web service specifications, we use the word "fault" whenever the specification also uses it, even if the word "failure" is more appropriate.

Looker et al. [18] analyze dependability of Web services by injecting faults in messages. They differentiate in physical faults, software faults, resource-management faults, communication faults and life-cycle faults. Gorbenko et al. [19] distinguish between errors in the "Network and service platform", in the "Web service software" and in the "Client software". Both works, however, do not consider all layers of the Web service stack as we do.

3 Fault Classification

The Web service platform architecture [1] categorizes the required middleware functions for facilitating interactions among the services of an SOA-based application in several layers. These layers are depicted Fig. 1 along with the Web service standards that specify the layer's functionality.

The *component layer* (Sect. 3.5) addresses the realization of an application's business logic which invokes the business functions the application is composed of, technologically rendered as Web services. One possible incarnation of a technology that is widely used for service orchestration in an SOA environment is the Web Service Business Process Execution Language (BPEL [2]), which hence is the focus of the discussion in the remainder of this paper. Composite applications often require nontrivial quality of service from the orchestrated services; typical examples of such nonfunctional component or service characteristics is reliability of interactions, transactional behavior of a set of services or security-related aspects, such as ensuring message integrity or message confidentiality. These functions are provided by the *quality of service* (Sect. 3.4) layer. Apart from nonfunctional properties, a (Web service) component is characterized by a *description* (Sect. 3.5) of its functional interface in form of a WSDL document [20], specifying the business functions supported by the service along with the message types they consume and produce. Concrete ordering of consumed and sent messages may be defined using BPEL. Requesting applications interact with service components by exchanging messages. Messages have a well-defined format that follows the SOAP specification; multiple messages may be interrelated to form potentially complex message

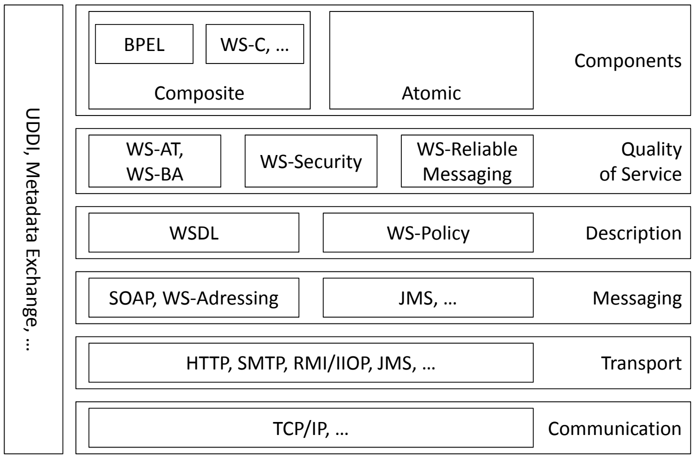

Fig. 1. Layers in the Web Service Platform Architecture, adapted from [1]

exchange patterns using the mechanisms provided by WS-Addressing [21] on the *messaging layer* (Sect. 3.3) such as means for identification of communicating entities and messages as well as message correlation. SOAP messages can be transmitted between components using different network transport protocols, depending on the requester's requirements, these are reflected by the *transport layer* (Sect. 3.2). The functions provided by the *communication layer* (Sect. 3.1) focus on the transmission of "raw data" among communication partners, potentially crossing the boundaries of one physical machine. Typically this functionality is provided by network transport protocols such as the Transmission Control Procotol (TCP) or the User Datagram Protocol (UDP), with themselves rely on lower level protocols such as the Internet Protocol (IP) for data transmission. In case the partners participating in the interaction reside on the same physical machine, machine-local data transmission mechanisms, such as shared memory, can be used on the communication layer (e. g. invoking a Web service implemented as an Enterprise Java Bean on the same machine). As Web services are defined as software systems that interact "over a network" [22] we focus on networked interactions in the remainder of this paper.

In the subsequent sections we discuss the fault handling behavior employed on each of the aforementioned layers in detail by describing a message flow between a service requester and service provider along with the faults that can occur on each layer and their respective fault handling strategies. For the following discussion we chose one concrete technology for the implementation of each layer: The application on the component layer is implemented using BPEL. Messages are exchanged reliably using WS-Reliable Messaging [23], encoded using SOAP and transported using HTTP over TCP/IP. Variants of these setting are briefly discussed in the respective sections. Figure 2 illustrates this setting. Each arc in the figure identifies a certain step in the overall message flow. During execution of the depicted interaction, messages are passed between the components implementing the individual layers of the application; message flow is thereby restricted

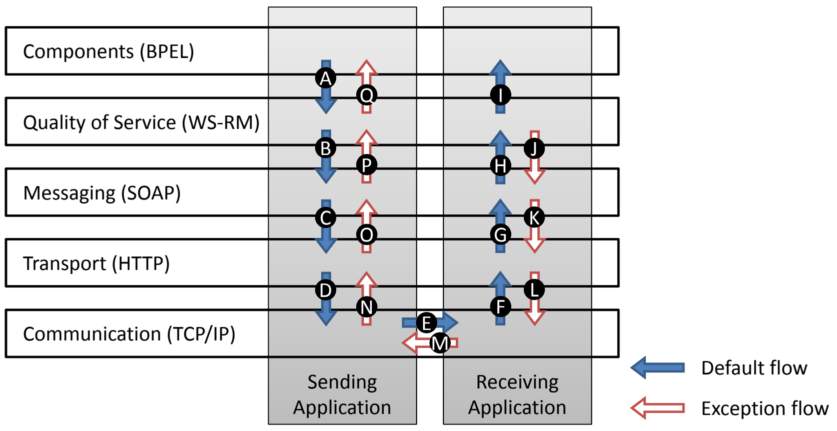

Fig. 2. Message Passing Through the Layers

to adjacent layers. In case a fault occurs it can be (i) either *handled* within the layer it occurred on or (ii) it can be *propagated* to a higher layer, which can then again decide to either handle the fault or propagate it to the next higher level. As the functionality provided by the description layer is not invoked during runtime of the depicted message exchange but during application build-time, this layer has been omitted from Fig. 2.

Note that faults occurring in the *runtime environment*, such as a component running out of memory during execution, errors due to bad memory and hard disk or system crashes are out of scope and hence not explicitly addressed in the discussion in the remainder of this paper. The same holds for faults resulting from *erroneous implementation*, i. e. implementations not conforming to their specification. Note that an implementation error may manifest in a fault in an arbitrary layer. Hence, such faults are out of scope of this paper and we focus on the specified behavior.

In the following sections we classify the faults according to the layers in the Web service stack. The individual layers are explained bottom-up from the communication layer to the component layer along with their respective fault types.

3.1 Communication Layer Faults

Generally, we subsume all protocols and mechanisms of the OSI Layered Network Model [24] below layer 7, i. e. the OSI application layer, as providing a platform for communicating data among the participants of an interaction and hence refer to them collectively as the communication layer of an SOA-based application.

The faults that may occur on this layer (triggered by a message flowing along arc D in Fig. 2) on the side of the sending application include *connectivity faults* where a sender cannot establish a connection with a receiver (arc E) or *data integrity faults* where the data exchanged between them is corrupted (arc E or arc M).

Examples for connectivity faults in applications are (i) problems in name resolution, i. e. a host name cannot be resolved to an IP address through DNS [25], (ii) problems during message routing from sender to receiver, i. e. host or network unreachable or

(iii) unavailability of a network endpoint, i. e. connection refused, due to the service provider not being ready to process incoming data. An example for a data integrity fault is loss of packets exchanged as part of an interaction.

Connectivity faults are typically not handled directly on the communication layer (e. g. by retrying a failed connection attempt at a later point in time) but are instead propagated up in the layered application architecture on the side of the sending application (arc N). If it is desired that these faults should be handled before reaching the requesting application's component layer, fault handling must be carried out on the quality of service layer as described in Sect. 3.4.

As data integrity faults may occur quite frequently—especially when using unreliable transports—some communication layer protocols, such as TCP, employ corresponding fault handling mechanisms directly on the communication layer, such as a retransmit of lost packets.

3.2 Transport Layer Faults

When a message is passed from the messaging layer down to the transport layer (arc C in Fig. 2) as part of the execution of the Web service runtime's binding implementation, the sending application encodes the message to be sent in a representation that can be conveyed using the chosen transport mechanism. In terms of the OSI layered model, this transport mechanism resides on layer seven, i. e. the application layer. Once the message has been encoded, the sending application uses the communication primitives provided by the chosen transport protocol to transmit the message to the receiving application—i. e. the component to be invoked—and potentially consumes response messages that may be sent by the service.

Each network transport protocol used on the transport layer may employ its own mechanisms for identifying and creating transport layer faults. In case of the Hypertext Transfer Protocol (HTTP) [26], transport layer faults are identified by an HTTP error code (e. g. "403 Forbidden" in case a requester is not allowed to access a certain resource exposed using HTTP). Note that—in contrast to communication layer faults—these faults do not occur on the side of the sending application, but at the receiving application and are then propagated back to the sender along the arcs L, M and N in Fig. 2. This is even the case for client-side error codes (4xx) indicating that the client made a wrong request (from the view of the server). Note that the communication along these arcs is a normal communication taking place in the established http connection. For the communication layer, the HTTP error is not an error. Communication layer faults propagated to the transport layer and transport layer faults are propagated to the next higher level (arc O in Fig. 2), i. e. the messaging layer.

3.3 Messaging Layer Faults

The messaging layer comprises the functionality of encoding messages coming through the quality of service layer from an application (along arc B in Fig. 2), processing the message according to the SOAP processing model [27], adding addressing information about the message's destination in form of corresponding WS-Addressing headers [21] and passing it on to the transport layer through the Web service runtime's binding implementation (arc C).

According to the SOAP specification, the following messaging layer fault types are distinguished: (i) a *VersionMismatch* fault identifies a fault due to an incompatible message format version; (ii) a *MustUnderstand* fault is generated when a receiver cannot process a mandatory SOAP header block; (iii) a *DataEncodingUnknown* is generated when a SOAP message uses an encoding that is not supported by a receiving SOAP node; (iv) a generic *Sender* fault represents invalid or missing message content as generated by the sending application and (v) a *Receiver* fault represents a fault that occurred due to (potentially transient) problems on the side of the receiving applications. In case of the latter fault type, resending the same SOAP envelope at a later point in time may result in successful server-side message processing. Similar to transport layer faults, these faults occur on the side of the receiving application and are propagated to the sender along the arcs K, L, M, N, O and P.

The mapping between messaging layer SOAP faults and transport layer fault status codes is defined as part of the specification of a Web service binding. In case of the SOAP/HTTP binding [28] all SOAP faults except *Sender* faults map to HTTP status code 500, which indicates a server-side processing error [26]. *Sender* faults map to a HTTP status code 400, which indicates a client-side error resulting in an invalid request message.

The SOAP processing model includes the definition of a routing concept from an initial sender over intermediary nodes to an ultimate receiver. This concept is one possible implementation of the enterprise integration patterns by Hohpe and Wolf [29]. Concrete examples are services for message encryption and message logging [30]. Messages may also be transmitted over different transports before reaching the ultimate receiver. For instance, SOAP/JMS may be used from the initial sender to the encryption service, the encryption service sends the encrypted message using SOAP/JMS to a messaging gateway. Finally, the messaging gateway uses SOAP/HTTP as the ultimate receiver only supports the SOAP/HTTP binding. As a consequence, a fault raised by an intermediary or the ultimate receiver may not just be propagated to the proceeding node which established a connection using a specific transport, but has to be routed to the initial sender. For that purpose, the WS-Addressing headers `ReplyTo` and `FaultTo` are defined. `ReplyTo` defines the endpoint reference, where the reply message should be directed. `FaultTo` is used to specify a different SOAP node to direct the fault to. In other words, the latter header is especially useful in multi-hop interactions, where a SOAP message is routed through several intermediaries, when only the original sender of the message should be notified about processing errors on any of the SOAP processing nodes along the message path.

Faults that may occur during message creation and processing on the side of the sending application are not rendered as SOAP faults but instead propagated to the component layer using the error handling mechanisms of the Web service runtime's implementation, e. g. Java exceptions when the runtime is implemented in the Java programming language. Note that this is not required by the SOAP specification, but typically implemented.

3.4 Quality of Service Layer Faults

The quality of service layer adds nonfunctional capabilities to Web services. These include support for transactions, security and reliable transfer of messages.

Transactions are implemented using the WS-Coordination framework, which in turn offers WS-AtomicTransactions (WS-AT) for interoperable two-phase commits and WS-BusinessActivity (WS-BA) for long-running compensation-based transactions [31]. WS-Security ensures message integrity and confidentiality [32]. In this paper, we focus on WS-ReliableMessaging (WS-RM) which enables reliable end-to-end messaging. That means even if SOAP intermediaries with different transports inbetween are used (see Sect. 3.3 for an example), the communication from the initial sender to the ultimate receiver is reliable. In case no component of the quality of service layer is used, messages from the component are directly passed to the messaging layer and messages from the messaging layer are directly passed to the component.

By using WS-RM, faults propagated from the messaging layer (arc P) are handled by the WS-RM component. WS-RM offers the configuration options AtLeastOnce, AtMostOnce, ExcactlyOnce and InOrder. WS-RM places messages in sequences. Each message takes a running number enabling in order delivery. Ranges of received messages are acknowledged by the receiver, which enables at least once and exactly once delivery. At most once delivery does not require acknowledgments as no delivery conforms to at most once. WS-RM defines faults which are propagated to the component layer. These faults indicate faults at the WS-RM processing itself, such as notifying that an endpoint is not WS-RM aware, an invalid acknowledgment is received or that the maximum value for numbering messages has been reached. They may be generated by the sending and the receiving quality of service layer. A sequence takes an expiry time. In case a sequence is not completed until the expiry, the sequence is terminated and a `SequenceTerminated` fault is raised through arc Q. The receiver may either discard the entire sequence, discard all messages following the first gap or discard nothing. A permanent fault at the client side (e. g. "403 Forbidden") is also propagated as SequenceTerminated fault to the sender application. A WS-RM implementation is not required to wait until the expiry is met and may propagate this fault earlier.

3.5 Component Layer Faults

Faults that may occur on the component layer, i. e. the layer on which an application's business logic resides, differ substantially from the fault types of the lower layers described so far.

As the latter are—under the given assumption of *absence* of erroneous implementations presented in Sect. 3—often transient errors that may be handled by retrying a message exchange after its failure, component layer faults are typically permanent in nature and an indication of an error in the application logic of a component that has technically been invoked successfully (by transmitting a request message along the arcs A to I and propagating the application fault by sending a new message). Component layer faults can be made more tangible by classifying them into two groups: faults reflected in the *component's interface description* or in the *component's implementation*.

Component Interface Description: WSDL. *Component interface faults* refer to faults which are specified as part of the functional component contract, i. e. the component's WSDL description (Fig. 1). These faults reflect an error situation in the application logic of a component and are hence—in contrast to the layers below the component layer—expected during application design time.

An example for a component interface fault is a calculator component, whose *divide* method signature defines a separate fault in addition to a request and a result message for notifying a requester when e. g. a division by zero or a range overflow occurs. Using this method, the requesting application gets notified about the component layer fault "out of band" of the regular response and can hence clearly identify the error situation and handle it accordingly.

In WSDL, component layer faults are specified through the `fault`-element of a WSDL-`operation` in WSDL 1.1 [20] or an `interface` in WSDL 2.0 [33] and typed using WSDL 2.0's typing mechanism. Whether a fault can be specified as part of the interface description of a component's operation is dependent on its *operation type* in WSDL 1.1 or its *message exchange pattern* in WSDL 2.0, respectively. In case an operation follows a *one-way/in-only* message exchange pattern, no faults may be propagated back to the sending application due to the "fire and forget" nature of the interaction. In the WSDL 2.0 specification this is referred to as *no faults propagation rule*. If this behavior is undesired, other message exchange patterns (e. g. *request-response/in-out*) should be chosen for the respective operation. In addition to these patterns, where a component is required to send a response even in case no fault occurs, WSDL 2.0 defines the *robust-in-only* message exchange pattern in which a fault may be propagated to the requesting application. The WSDL specification, however, lacks a clear description how requesting applications are supposed to handle these optional faults on a technical level, e. g. how long to wait for a fault for a particular component invocation until it is assumed that the invocation was successful. Thus, this pattern is underspecified [34].

Component Implementation: BPEL. A component interface fault is created during the execution of the implementation of a component's application logic. As not all faults that may occur during execution of a component should become visible outside the component itself, component implementation faults can be—similar to the other fault types—be either handled or propagated. Propagation of component layer faults typically result in involving a human in the fault handling process.

For component layer-internal fault handling, BPEL defines the concept of fault handlers which can be attached to either `scope` or `invoke` activities in which process modelers may specify application logic to be executed when a fault occurs. A fault can be created either explicitly as part of the process model through the `throw` activity, or can be raised implicitly during execution of e. g. an `invoke` interaction which results in a fault declared as part of a component's interface (cf. Sect. 3.5).

Although the BPEL specification defines how to react to interface layer faults, no explicit provisions are made how to treat faults that occur on layers below the component layer during the process of sending an invocation request to a component (i. e. when executing arcs A to E). Hence, implementations differ in their behavior with regard to handling such faults. A common way to treat transient transport layer faults, which is e. g. used in *Apache ODE*[1], is distinguishing business *faults* and technical *failures*. Whereas business faults are propagated into the process for "regular" fault handling through the processes fault handlers, failures result in suspending process instance execution and notification of an administrator who—after resolving the problem—may

[1] `http://ode.apache.org`

resume process instance execution. Another approach to handling lower layer faults is to propagate such faults to the process in form of custom typed BPEL faults which may then be handled using BPEL's fault handling mechanisms. The approach is applied in *ActiveBPEL*[2]. Apache ODE may be configured to this behavior by using the `faultOnFailure` attribute.

4 Implemented Fault Propagation

This section presents a concrete way to implement the presented exception handling strategies. We use Apache ODE 1.3.8, Apache Sandesha2[3] and Apache Axis2 1.3.1[4] to illustrate the implementation concepts. These implementations are put to relation to the layers of the Web service stack (Fig. 1) in Fig. 4: Being a BPEL engine, Apache ODE implements a component. Apache ODE calls Axis which implements the messaging and the transport layer. Sandesha is a plugin for Axis implementing WS-RM, which we focus on. Axis uses HTTP components to implement the transport layer which in turn uses `java.io` from the Java runtime as implementation for the runtime layer.

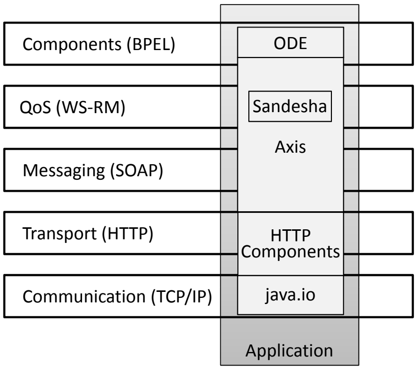

Fig. 3. Layer Implementations

4.1 Apache ODE

Apache ODE (ODE for short) is a BPEL engine developed by the Apache Software foundation. After a process is instantiated, a `BpelRuntimeContext` is available. In case of an invoke activity, its `invoke` method is called to invoke a service. This call reaches the `SoapExternalService` class. Depending on the type of the invoke, a `OutOnlyAxisOperationClient` or an `OutInAxisOperationClient` (cf. Sect. 4.2) is created. In the case of a one-way invoke, an exception raised by the `OutOnlyAxis-OperationClient` is logged without propagating into the process. In case of a two-way invoke, an exception raised by the `OutInAxisOperationClient` the exception is put as failure in a `PartnerRoleMessageExchange` object `odeMex`. If a SOAP fault is received as reply, this fault is put as fault in `odeMex`. Otherwise, the reply message is put there. An exception on the handling of the reply message is rendered as fault. Received faults and locally generated faults and failures are propagated to the parent activity by calling the `completed` or `failure` method. The `ACTIVITYGUARD` class implements the failure method. Here, the `faultOnFailure` property is checked and the failure either converted to a fault or an administrator involved. Faults are propagated up the activity hierarchy (using the `completed` method) until a scope is reached. Here, a failure

[2] `http://www.activebpel.org`

[3] `http://ws.apache.org/sandesha/sandesha2/`

[4] `http://ws.apache.org/axis2`

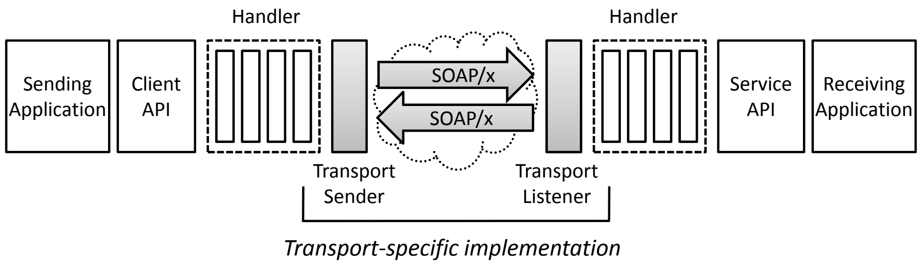

Fig. 4. Axis 2 Architecture

is converted to a fault and the `completed` method is called. Subsequently, the usual BPEL fault handling runs as described in the BPEL specification [2,35,36]. In short, all activities in the scope the activity belongs to are terminated and a fault handler of the scope is called. Here, completed activities of the scope may be compensated by calling compensation activities. A fault may be re-thrown to the parent scope, where the same handling starts.

4.2 Apache Axis2

Apache Axis 2 (Axis for short) is a Web service runtime that supports different Web service specifications and transport protocols. In Fig. 4, a high-level overview of Axis' architecture[5] is depicted.

Axis supports two different service invocation styles: client may (i) either use build-time tools to generate a client-side proxy object based on the service's WSDL description which wraps the interaction with the service provider or (ii) use Axis' `ServiceClient` API to dynamically construct the service invocation request at run-time. For this purpose, the API exposes several methods for SOAP message generation (e. g. `addHeader`, `setTargetEPR`) and sending (e. g. `fireAndForget`, `sendReceive`, `sendReceiveNonBlocking`, `sendRobust`). These methods can be used to implement different message exchange patterns such as *in-only* in case of `fireAndForget` or synchronous or asynchronous *in-out* in case of `sendReceive` and `sendReceiveNon-Blocking`, respectively.

The following description of the internal actions carried out by the Axis run-time during execution of a Web service interaction assumes that a client uses the `ServiceClient` API directly. As generated proxy objects rely on the `ServiceClient` API internally as well, there is no difference in fault handling behavior on the layers below the component layer when using either service invocation style. The invocation styles, however, differ with regard to their component layer fault handling behavior. Whereas the proxy objects generated during build-time render interface faults as typed faults in the client's programming language (e. g. an exception in Java), the `ServiceClient` API propagates them to the component layer in form of a generic `AxisFault`.

[5] Adapted from `http://ws.apache.org/axis2/1_4_1/userguide.html`

Once a client application has called an operation of the `ServiceClient` object, a `OperationClient` object is created that corresponds to the message exchange pattern implemented by the invoked `ServiceClient` method. In case the `fireAndForget` method of the `ServiceClient` is invoked, which implements an *in-only* WSDL message exchange pattern, a `OutOnlyAxisOperationClient` object is created, which furthermore contains the client-defined SOAP envelope and determines the transport protocol to be used to invoke the service based on the client-defined service endpoint address in form of a `TransportOutDescriptionObject`.

After creation of the request message, it is passed into the `AxisEngine` in form of a `MessageContext` object. The engine subsequently passes the message context object to several configurable handler objects (cf. Fig. 4) which may perform additional processing steps before the message is passed to the implementation of the `TransportSender` interface corresponding to the transport protocol that should be used for the respective interaction. The `TransportSender` implementation for using a HTTP transport is the `CommonsHTTPTransportSender`. Additional processing steps include WS-Security and WS-RM steps. In the case of WS-RM, the `SandeshaOutHandler` is used. Here, the WS-RM data are created or fetched from an internal storage (e. g. sequence number) and added to the message[6]. The message is put to a `SenderBeanManager`, where messages to send are persistently stored to have them available for a retransmit. The transport sender serializes the message payload in the `MessageContext` and sends it to the receiver using the `CommonsHTTPClient` `sendViaPost` method. The message transmission is carried out using the *Apache HttpComponents Client*[7].

Any exceptions occurring during message creation and processing within Axis (including the WS-RM phase) at the side of the message sender as described before result in the creation of an `AxisFault` which is propagated back to the calling client application. The reason of the fault (e. g. a `MailformedURLException` in case of an invalid URL in the defined endpoint address or an `IOException` in case of an unavailable receiver) is embedded in the generated `AxisFault` which allows clients to individually handle different fault types.

On the server-side Axis implementation, the request is received either through the `AxisServlet` or a stand-alone `HTTPTransportReceiver`. The SOAP request message is extracted from the incoming HTTP request and stored into a `MessageContext` object which is then passed through the handlers of the server-side Axis runtime (see Fig. 4) to the service's application logic. Any fault occurring on the way up to the component layer is rendered as a SOAP fault with its fault code chosen according to the description in Sect. 3.3. The WS-RM part checks if the message is a WS-RM message or if there are WS-RM headers present and executes the respective logic. For instance, in the case of the `SequenceAcknowledgement` header, acknowledged message numbers are compared with the numbers of the sent messages. In case a message is acknowledged, it is removed from the `SenderBeanManager`. The unacknowledged messages are sent using the `SandeshaThread`, which in turn periodically queries the sender bean manager.

[6] Details on the internal processing by Sandesha2 is provided by `http://ws.apache.org/sandesha/sandesha2/architectureGuide.html`

[7] `http://hc.apache.org/`

After message transmission, the client-side transport sender interprets the status of the message submission based on the received HTTP status code (cf. mapping between SOAP faults and HTTP status codes in Sect. 3.2). In case the receiving side signals an HTTP code of 2xx, the message transmission is considered successful and the payload of the response message (which e. g. in case of an *in-out* message exchange pattern contains the result of the service invocation) is extracted and stored into a `MessageContext` object. In case an HTTP status code of 4xx (client error) or 5xx (server error) is received, the response message is checked for the presence of a SOAP fault which is extracted and stored into a `MessageContext` object similar to the response case.

Whether the response or fault is propagated back to the calling application through Axis' handlers is dependent on the concrete message exchange pattern of the executed operation. If the call is an *in-only* operation, the response and error messages are discarded and not processed further by Axis. If the call on the other hand is a *in-out* operation (realized by an `OutInAxisOperationClient`), either the created message context object is passed back to the sending client application through Axis' handlers or an exception corresponding to the occurred fault is thrown and propagated back, either directly in case of a blocking invocation of `sendReceive` or by invoking a client-provided callback in case of `sendReceiveNonBlocking`. Processing the response status codes even in case of a `fireAndForget` invocation allows for developing implementations of the `TransportSender` interface, which are independent of any concrete message exchange patterns.

Alternative binding implementations, such as the `MailTransport`, which are realized as further implementations of the `TransportSender` interface, generally follow the same approach to identifying, wrapping and propagating faults back to the calling client application.

5 Conclusions and Future Work

This paper provided an overview on the fault handling on all layers in the Web service stack. Special emphasis was put on the analysis of the interplay between the different layers. We showed that current related work in the field does not holistically regard the communication between different services. This paper helps to foster the awareness of the different layers of the Web service stack and enables a more detailed analysis of impacts of new solutions to the Web service stack.

Not all specifications cover fault handling completely. The fault-handling-mechanisms defined e. g. as part of the BPEL specification alone are not sufficient to enable a robust behavior of a business process in all cases: BPEL engine implementations differ e. g. in the way how they handle a fault occurring on the communication layer. Vendors include custom extensions; e. g. ODE suspends a process and allows an administrator to decide how to handle the situation. Thus, guidelines are needed to enable developing applications which are reaching a consistent state even in the case of a failure during the execution.

Currently, faults are handled by a component in each layer. In our future work, we plan to use BPEL to coordinate fault handling in the quality of service layer and below by extending the work presented in [30].

In this work, we discussed the implementations of Apache ODE and Apache Axis only. Future work has to investigate the behavior of other WS runtimes such as the Active BPEL engine and Apache CXF.

Current formalizations of BPEL and other process calculi currently do not capture the behavior of the Web service stack. Thus, our future work is to use our findings to include the behavior of the middleware in verification of processes.

References

1. Curbera, F., Leymann, F., Storey, T., Ferguson, D., Weerawarana, S.: Web Services Platform Architecture: SOAP, WSDL, WS Policy, WS Addressing, WS BPEL, WS-Reliable Messaging and More. Prentice Hall PTR, Englewood Cliffs (2005)
2. OASIS: Web Services Business Process Execution Language Version 2.0 – OASIS Standard (2007)
3. Papazoglou, M.P.: Web Services: Principles and Technology. Prentice-Hall, Englewood Cliffs (2007)
4. Coulouris, G., Dollimore, J., Kindberg, T.: Distributed Systems: Concepts and Design. In: Distributed Systems: Concepts and Design, 4th edn. (2005)
5. Jiajia, W., Junliang, C., Yong, P., Meng, X.: A Multi-Policy Exception Handling System for BPEL Processes. In: First International Conference on Communications and Networking in, China (2006)
6. Modafferi, S., Mussi, E., Pernici, B.: SH-BPEL: a Self-Healing Plug-In for WS-BPEL Engines. In: 1st Workshop on Middleware for Service Oriented Computing, ACM, New York (2006)
7. Guidi, C., Lanese, I., Montesi, F., Zavattaro, G.: On the Interplay Between Fault Handling and Request-Response Service Invocations. In: 8th International Conference on Application of Concurrency to System Design ACSD (2008)
8. Ardissono, L., Furnari, R., Goy, A., Petrone, G., Segnan, M.: Fault tolerant web service orchestration by means of diagnosis. In: Gruhn, V., Oquendo, F. (eds.) EWSA 2006. LNCS, vol. 4344, pp. 2–16. Springer, Heidelberg (2006)
9. Friedrich, G., Fugini, M., Mussi, E., Pernici, B., Tagni, G.: Exception Handling for Repair in Service-Based Processes. IEEE Transactions on Software Engineering 99(2), 198–215 (2010)
10. Console, L., et al.: WS-DIAMOND: Web Services-DIAgnosability, MONitoring, and Diagnosis. In: At Your Service: Service-Oriented Computing from an EU Perspective, pp. 213–239. MIT Press, Cambridge (June 2009)
11. Russell, N., van der Aalst, W., ter Hofstede, A.: Workflow Exception Patterns. In: Advanced Information Systems Engineering. In: Dubois, E., Pohl, K. (eds.) CAiSE 2006. LNCS, vol. 4001, pp. 288–302. Springer, Heidelberg (2006)
12. Lohmann, N.: Communication models for services. In: 2nd Central-European Workshop on Services and their Composition (ZEUS 2010), vol. 563, CEUR (2010)
13. Lohmann, N.: A Feature-Complete Petri Net Semantics for WS-BPEL 2.0. In: Dumas, M., Heckel, R. (eds.) WS-FM 2007. LNCS, vol. 4937, pp. 77–91. Springer, Heidelberg (2008)
14. Eder, J., Liebhart, W.: The Workflow Activity Model WAMO. In: Conference on Cooperative Information Systems (CoopIs), pp. 87–98 (1995)
15. Eder, J., Liebhart, W.: Workflow Recovery. In: COOPIS 1996: First International Conference on Cooperative Information Systems, IEEE Computer Society, Los Alamitos (1996)
16. Mourao, H., Antunes, P.: Supporting Effective Unexpected Exceptions Handling in Workflow Management Systems. In: SAC 2007, pp. 1242–1249. ACM, New York (2007)

17. Avižienis, A., Randell, B., Landwehr, C.: Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Transactions on Dependable and Secure Computing 1(1), 11–33 (2004)
18. Looker, N., Munro, M., Xu, J.: Simulating errors in web services. International Journal of Simulation Systems 5(5), 29–37 (2004)
19. Gorbenko, A., Romanovsky, A., Kharchenko, V., Mikhaylichenko, A.: Experimenting with Exception Propagation Mechanisms in Service-Oriented Architecture. In: WEH 2008: 4th International Workshop on Exception Handling, ACM, New York (2008)
20. Christensen, E., Crubera, F., Meredith, G., Weerawarana, S.: Web Services Description Language (WSDL) 1.1. W3C Note (2001), `http://www.w3.org/TR/wsdl`
21. Gudgin, M., Hadley, M., Rogers, T.: Web Services Addressing 1.0 – Core. W3C Recommendation (2006), `http://www.w3.org/TR/ws-addr-core/`
22. Haas, H., Booth, D., Newcomer, E., Champion, M., Orchard, D., Ferris, C., McCabe, F.: Web Services Architecture. W3C Working Group Note (2004), `http://www.w3.org/TR/ws-arch/`
23. OASIS: Web Services Reliable Messaging (WS-ReliableMessaging) Version 1.2. (2009)
24. ISO 7498-1:1994: Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model. ISO, Geneva, Switzerland
25. Mockapetris, P.V.: Domain Names - Implementation and Specification. RFC 1035 (1987)
26. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (1999)
27. Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J.J., Nielsen, H.F., Karmarkar, A., Lafon, Y.: SOAP Version 1.2 Part 1: Messaging Framework. W3C Recommendation (2007), `http://www.w3.org/TR/soap12-part1/`
28. Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J.J., Nielsen, H.F., Karmarkar, A., Lafon, Y.: SOAP Version 1.2 Part 2: Adjuncts. W3C Recommendation (2007), `http://www.w3.org/TR/soap12-part2/`
29. Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Professional, Reading (2003)
30. Scheibler, T., Karastoyanova, D., Leymann, F.: Dynamic Message Routing Using Processes. In: KiVS 2009, Springer, Heidelberg (March 2009)
31. OASIS: OASIS Web Services Transaction (WS-TX) TC (2009), `http://www.oasis-open.org/committees/ws-tx/`
32. OASIS: Web Services Security: SOAP Message Security 1.1 (2006), `http://docs.oasis-open.org/wss/v1.1/`
33. Chinnici, R., Gudgin, M., Moreau, J.J., Schlimmer, J., Weerawarana, S.: Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. W3C Recommendation (2004), `http://www.w3.org/TR/wsdl20/`
34. Nitzsche, J., van Lessen, T., Leymann, F.: WSDL 2.0 Message Exchange Patterns: Limitations and Opportunities. In: 3rd International Conference on Internet and Web Applications and Services (ICIW), IEEE Computer Society, Los Alamitos (June 2008)
35. Curbera, F., Khalaf, R., Leymann, F., Weerawarana, S.: Exception Handling in the BPEL4WS Language. In: van der Aalst, W.M.P., ter Hofstede, A.H.M., Weske, M. (eds.) BPM 2003. LNCS, vol. 2678, pp. 276–290. Springer, Heidelberg (2003)
36. Khalaf, R., Roller, D., Leymann, F.: Revisiting the Behavior of Fault and Compensation Handlers in WS-BPEL. In: Meersman, R., Dillon, T., Herrero, P. (eds.) OTM 2009. LNCS, vol. 5870, pp. 286–303. Springer, Heidelberg (2009)