Fault Handling Across the Web Services Stack

Pablo Fuentetaja Abad

Course of Study: Computer Science

Examiner: Prof. Dr. Frank Leymann
Supervisor: Dipl.-Inf. Oliver Kopp
Commenced: January 14, 2008
Completed: July 15, 2008

CR-Classification: C.2.0, C.2.1, C.2.2, C.2.4, D.2.11
Acknowledgments

To my family and friends. Their support has helped me during the difficult moments. Without them this would have not been possible.

To Prof. Dr. F. Leymann. For offer me the opportunity of developing this work in the Institut für Architektur von Anwendungssystemen.

To Dipl.-Inf. O. Kopp. For his work, help, and supervision during these months.

To Dipl.-Inf. Matthias Wieland. For his last supervision and comments.
Abstract

The Business Process Execution Language (BPEL) is an XML based language for describing business process behaviour based on Web services. The BPEL notation includes flow control, variables, concurrent execution, input and output, transaction scoping/compensation, and error handling. These processes are executed on a BPEL engine which calls and receives messages from external parties. The BPEL process is suspended or terminated if such communication fails, not providing any detailed information about the cause.

The aim of this diploma thesis is the description of the different communication faults that can be found throughout the Web Services Stack, how they are reflected and describe a general concept of fault handling.

Two BPEL engines are use on this thesis “Oracle BPEL Process Manager” and “Apache ODE”. The first one provides some specific standard faults than are able to manage communication faults. This means are taken as a reference to develop a concept which will be extrapolated into the second engine.
# Table of Contents

1. INTRODUCTION ........................................................................................................... 9

2. BACKGROUND .............................................................................................................. 11
   2.1. OSI MODEL AND TCP/IP MODEL ...................................................................... 11
      2.1.1 Description of OSI Layers ......................................................................... 12
      2.1.2 Communication between layers ................................................................. 14
   2.2. WEB SERVICES ..................................................................................................... 15
      2.2.1 Web Services Protocol Stack ..................................................................... 16
   2.3 OSI MODEL AND WEB SERVICE STACK .......................................................... 18

3. MOTIVATING EXAMPLE .............................................................................................. 21
   3.1 ERROR SENDING IP DATAGRAMS .................................................................. 22
      3.1.1 Destination Unreachable Example ................................................................. 23
   3.2 ERROR SENDING TCP SEGMENTS .................................................................... 24
      3.2.1 TCP Segment Lost ..................................................................................... 24
   3.3 ERROR IN APPLICATION LAYER ........................................................................ 25
      3.3.1 HTTP Service Unavailable ....................................................................... 26
   3.4 ERROR IN MESSAGING - SOAP ...................................................................... 27
      3.4.1 SOAP mustUnderstand .............................................................................. 29

4. FAULT HANDLING IN THE DIFFERENT LAYERS ...................................................... 31
   4.1 FAULT HANDLING IN TCP/IP .......................................................................... 31
      4.1.1 Fault Handling in IP .................................................................................... 31
      4.1.2 Fault Handling in TCP ............................................................................... 34
   4.2 FAULT HANDLING IN HTTP ............................................................................. 41
      4.2.1 HTTP Message ........................................................................................... 42
      4.2.2 HTTP Status Response .............................................................................. 42
      4.2.3 HTTP Example ........................................................................................... 44
   4.3 FAULT HANDLING IN SOAP, ............................................................................ 45
      4.3.1 SOAP Syntax .............................................................................................. 46
      4.3.2 SOAP Processing Model ........................................................................... 48
      4.3.3 SOAP Protocol Binding ............................................................................. 51
      4.3.4 SOAP Examples ......................................................................................... 52
   4.4 FAULT HANDLING SPECIFIED IN WSDL ....................................................... 54
      4.4.1 SOAP Faults in WSDL .............................................................................. 54
   4.5 FAULT HANDLING IN BPEL ............................................................................. 57
      4.5.1 Business Faults .......................................................................................... 58
      4.5.2 Run-time Faults .......................................................................................... 58
   4.6. LAYER’S FAULTS ............................................................................................. 59
      4.6.1 Transport .................................................................................................... 59
      4.6.2 Application (HTTP) .................................................................................... 61
      4.6.3 Messaging Layer ........................................................................................ 62
      4.6.4 Description and Discovery ........................................................................ 64
      4.6.5 Business Processes Layer .......................................................................... 64
1. Introduction

BPEL (Business Process Execution Language) is a language to express executable business processes, called workflows. BPEL processes are executed on a BPEL engine. The engine calls external parties and receives messages from them. If communication fails, the BPEL process is not notified, but suspended or terminated.

Communication failures can happen throughout the Web Services stack [5, 6]: at the transport level, the messaging levels or at the process level. Until now, no overall concept exists on how to map failures from one level to the upper ones.

The goals of this thesis are to describe the different faults that can be raised across the Web Services stack and how they are reflected; describe the general concept of fault handling across the Web Services stack and illustrate how some two BPEL engines deal with this kind of faults.

It is important for the correct understanding of this thesis to have in mind the following concepts:

- **Failure**: the inability of a system to perform its required functions within specified performance requirements. Applied to the thesis, a failure in a business process would be the loss of communication between two of the servers hosting services for a workflow.

- **Fault**: abnormal condition that may cause the fail of a functional unit to perform a required function. Deviation from a desired or intended state. Applied to the thesis an example of fault can be when a web service operation cannot complete successfully.

- **Error**: discrepancy between a condition and the true, specified or theoretically correct value or condition.

This thesis is structured in different chapters. The thesis is intended to be a step by step study to reach the goals. From the first documentation chapter, to build the basic knowledge and the motivation of the thesis. To the latest ones, oriented to show how difficult is to deal with communication fault within the Web Services stack. All this is supported by a practical example (Travel Agency) along the thesis.

Next, a short description of what can be found in each of the chapters:

- **Background**: introduction to the technological background needed to understand the thesis. Brief introduction to the OSI Model and to Web Services, including brief description of their different layers.

- **Motivating Example**: for an better understanding of all that will be explained an example is needed. With the help of this example it can be demonstrated the way fault handling works. The example consists of a customer (client) who wants to reserve a room for his holydays. The customer contacts a travel agency
(BPEL process) providing all the necessary information to complete the reservation. The travel agency with all the information received by the customer tries to make a reservation for a hotel room (Application Server).

- **Fault Handling in the different layers:** This chapter will go through the different layers identified in the background chapter, describing the different layer's mechanisms to deal with errors and failures that may occur in them, and how these faults are reflected. Also summarizes and describes the faults that can be arising within the corresponding layer in the Web Services stack, beginning from Transport Layer (TCP/IP) and ending with the Business Processes Layer (BPEL processes).

- **Motivating Example Extended:** in this chapter a BPEL <faultHandler> is added to our motivating example in order to <catch> the possible faults occurred during the communication with a Web Application.

- **Current Situation:** Illustrates how faults can be handled at the moment based on two relevant BPEL engines: Oracle BPEL Process Manager and Apache ODE.

- **Proposed Solution:** As a result of the thesis this chapter describes, based upon the motivating example presented in previous chapters, how the engines deal with the communication faults and defines the concept of how they could handle the different communication faults detailed in chapter 4.
2. Background

In this chapter we introduce the necessary basics to understand the main part of this diploma thesis. First, the concepts of OSI Model (section 2.1) and Web Services (section 2.2) are explained, including a brief description of its different layers. After that, the most important layers in the OSI Model are described in more detail in chapter 4.

2.1. OSI Model and TCP/IP Model

The Open Systems Interconnection Basic Reference Model (OSI Model for short) [1, 2, 3, 4] is a layered, abstract description for communications and computer network protocol design.

It is assumed that the reader is familiar with the concepts of TCP/IP and for this reason the OSI model is described in relation to it.

The OSI model has a total of seven layers while the TCP/IP model has only five layers (these layers have the equivalent one on the other model, see Figure 2.1.1). From top to bottom, the OSI Model consists of the Application, Presentation, Session, Transport, Network, Data Link, and Physical layers.

![OSI Model vs. TCP/IP Model](image)

Figure 2.1.1. OSI Model stack vs. TCP/IP stack.
2. Background

These layers (in both the OSI and the TCP/IP models) are identical up to the Network layer. From Transport layer the two models are not quite the same.

Each layer is a collection of related functions. The layers receive service from the layer below and provide services to the layer above it.

2.1.1 Description of OSI Layers

2.1.1.1 Physical Layer

The physical layer is the first level in the seven-level OSI model as well as in the five-layer TCP/IP reference model. This layer performs services requested by the layer above, data link layer.

The physical layer is the responsible of access to physical interconnection. Therefore, defines the electrical characteristics (voltage levels or tension between the wires…), mechanical (shape and design of the connectors, provision of the pins,…) and logical (signals exchanged). With a reception-transmission device (for example, a modem) is accessed to the physical environment and allow transmitting raw bits between two contiguous entities. Therefore, at this level is no software entities and, therefore, there is no protocol to the physical level.

2.1.1.2 Data Link layer

The data link layer is the second layer of the seven-layer OSI model as well as of the five-layer TCP/IP reference model. This layer issues service requests to the physical layer and responds to service requests from the network layer.

The data link layer is the level responsible for the data exchange between adjacent network nodes. Also, as in the transport and network levels, this level can provide reliable service (connection-oriented) or unreliable (not connection-oriented).

If the service is reliable (typical service in OSI computer network), a physical inspection of errors generated by the physical environment (detection and retrieval of data that have changed in a bit) and logical (detection and recovering lost data, disordered or duplicated) is done.

Also, there is a flow control between the two entities to prevent that an entity transmits faster than the other is capable of storing and processing.
2.1.1.3 Network Layer

The network layer is the third layer of seven-layer OSI model and also the third layer of five in the TCP/IP model. In the TCP/IP reference model it is called the Internet layer. The network layer issues service requests to the data link layer and responds to service requests from the transport layer.

The network layer is responsible for end to end packet routing and delivery. Each package or unit network data contains a header information control, including among other information, the direction of the target machine for the package in question. Based on this direction, each network entity, take a decision of routing to the final system. Also, as in the transport layer, network level can offer reliable service (connection-oriented) or unreliable.

2.1.1.4 Transport Layer

The data transport layer is the fourth layer of the seven-layer OSI model as well as of the five-layer TCP/IP reference model.

The transport layer is the responsible for transporting data between entities. If the service offered is reliable, the transport layer is responsible for the reliability of communication. Therefore, there is an entire control of physical and logical errors. Also, there is a flow control between the two entities to prevent that one entity transmit faster than the other is capable of storing and processing.

2.1.1.5 Session Layer

The session layer is fifth layer of the seven-level OSI model. It issues service requests to the transport layer and responds to service requests from the presentation layer.

The Session layer is the level responsible for managing and synchronizing dialogue between presentation level entities (coordinator of communication between systems). It provides for either full duplex or half-duplex operation and establishes checkpointing, adjournment, termination, and restart procedures. Several protocols exist at the session layer, including Remote Procedure Calls (RPCs) for example.

2.1.1.6 Presentation Layer

The presentation layer is the sixth level of the seven-layer OSI model. It issues service requests to the session layer and responds to service requests from the application layer.
The presentation layer is the level responsible for the syntax of information exchanged between implementation level entities (format the data is presented in). It is the level which deals with the presentation of the data exchanged by the application processes.

2.1.1.7 Application layer

The application layer is the top level of the seven-layer OSI model. It interfaces directly to and performs common application services for the application processes; it also issues requests to the presentation layer. Therefore different applications or user processes are running at this level.

2.1.2 Communication between layers

As has been described, OSI model has 7 layers. These layers are communicating with each other by passing units of data via SAP (Service Access Point).

The Service Access Point is an identifier of the software entity on the upper level, when at this level there is more than one entity. A SAP level "N" always identifies an entity level "N +1". Therefore, in the "bottom-up" calls, SAP are used to identify an entity that are at the next upper level. In the "top-down" calls always interacts with the same entity in the level immediately below. When a call is made from level "N +1" to level "N", the SAP of the remote entity "N +1" is passed as an argument over the call. The SAP represents the border or dividing line that defines the interface between adjacent levels of communications in a single system.

The unit of data at Layer N is called a PDU (Protocol Data Unit), and the unit of data at the Layer above that (Layer N+1) is called an SDU (Service Data Unit).

A SDU consists of layer N header information and an encapsulated message from layer N+1, which is called both the layer N SDU and the layer N+1 PDU.
When a message is passed down (e.g. HTTP → TCP), the lower layer adds a header to it. This is called encapsulation. The successive lower layer adds its own header to the received data from the layer above it (encapsulate).

In case that a message is moving from the bottom-up (e.g. TCP → HTTP), the message gets smaller. A message, first stripped off its header, then the inner contents is passed up. This is called "decapsulation". The successive upper layer receives the data message from the layer below, and then strips off its own header and passes the data up.

### 2.2. Web Services

The Web Service Stack (Figure 2.2.1) is a collection of protocols and standards that are used to exchange data between applications. The Web Services Stack is a collection of protocols for computer networks that are used to identify, locate, and to implement a Web Service interacts with another [5, 6, 7].
Different software applications developed in different programming languages, and executed on any platform, can use Web Services to exchange data in computer networks such as the Internet. Interoperability is achieved through the adoption of open standards (XML, SOAP, WSDL…).

Web Services can be standalone or linked together to provide enhanced functionality.

### 2.2.1 Web Services Protocol Stack

The following sections describe the main characteristics and primary functions performed at each of these layers.

![Web Services stack](image)

**Figure 2.2.1. Web Services stack [26]**

#### 2.2.1.1 Transport

The transport layer is the responsible of dealing with the various application-level transports that can be used to carry a message from one location to another.

Web Services can be based on different transport mechanisms. This includes protocols layered on the fifth or sixth OSI layer such as SMTP, HTTP, FTP, JMS, etc.
2.2.1.2 Messaging

The messaging layer is the responsible of encoding the Web Service messages for transport. How to format messages properly is defined in the messaging specification.

Simple Object Access Protocol (SOAP for short, [5, 8, 9, 10, 11]) is a specification that works on any protocol (HTTP generally). It is used to define an envelope to send messages from one end-point to another (SOAP defines the standard encoding for representing payload).

At the same time, SOAP is also routing protocol, provides a distributed processing model. At an initial SOAP sender is originated a SOAP message and is sent to an ultimate SOAP receiver via zero or more SOAP intermediaries.

WSDL ([5, 11, 13]) is a description language that describes the public interface of Web Services (See section 2.2.1.3). Thanks to WSDL binding, other protocols and services can be used to do messaging, instead of XML, such for example POJO/JMS or ASCII/SMTP.

WS-Addressing

WS-Addressing provides a standard mechanism to identify Web Services and Web Services messages regardless of the transport protocol that is used. This is reached by defining a standard to incorporate message addressing information into Web Services messages. [5, 11, 12]. WS-Addressing also provides an addressing method for SOAP messages.

2.2.1.3 Description and Discovery

This layer defines a set of services supporting the description and discovery of businesses, organizations, and other Web Services providers.

WSDL

Web Services Description Language (WSDL for short [5, 11, 13]) describes the public interface of Web Services. It is based on XML and describes the form of communication, the requirements of the protocol and formats the messages needed to interact with the services listed in its catalogue. The operations and messages that supports are described in abstract and then binding to the specific network protocol and format of the message.
2. Background

**UDDI**

The Universal Description and Discovery Interface specification (UDDI for short, [5, 11]) defines a metadata aggregation service. Service discovery relies on service publication; if a Web Service cannot be found or discovered if is not or cannot be published. The service client can make the service description available to an application at runtime.

2.2.1.4 Orchestration (Business Process)

This refers to the service logic that manages the composition of underlying services and achieves a new set of functionalities. Orchestration provides sequencing, message format translations, and state management as required in a collaboration scenario.

BPEL language provides the required constructs to build business processes based on orchestrations of services.

In other hand, an orchestration description denotes the order of Web Service invocations between two, or more, business partners to complete a multi-step business interaction from the viewpoint of one business partner.

A business process specifies the potential execution order of operations from a collection of Web Services, the data shared between these Web Services, which partners are involved and how they are involved in the business process, joint exception handling for collections of Web Services, and other issues involving how multiple services and organizations participate. BPEL specifies business processes and how they relate to Web Services.

2.3 OSI Model and Web Service Stack

The OSI model defined seven layers in which the application layer is viewed as a single layer. With the evolution to Web Service, the application layer is becoming increasingly more distributed and requires the definition of its own stack.

While the stack for a single service has been clearly defined, the stack that is dealing with multiple services is often limited to orchestration that does not necessarily cover all the attributes that are required to handle the aggregation of services based on collaboration.

To cope with the dynamic nature of the service network environment and to differentiate the aggregation of services from the behaviour of that aggregation, a collaboration layer must be added to the OSI stack.
The following figure (Figure 2.3.1) shows how the upper layers of OSI stack are divided into multiple sub-layers. The lower three sub-layers deal with a single service, while the upper two are concerned with service collaboration between multiple services.

The Transport and Session layers from the conventional OSI reference model can be grouped into the Transport layer in the Web Service stack.

OSI Presentation layer corresponds to the Messaging layer in the Web Service stack.

The Application layer from the conventional OSI reference model can be divided into the following sub-layers:

- Quality of Services
- Description and Discovery
- Orchestration (Business Process)
3. Motivating Example

For an easy understanding of all that was explained in the previous chapter, an example is needed. With the help of such an example it can be demonstrated the way fault handling works.

In our example, the customer (client) wants to book a room for his holydays. The customer puts in contact with a travel agency providing all the necessary information for the reservation. The travel agency with all the information received by the customer tries to make a reservation for a hotel room (see Figure 3.1.1). The following sections will describe the different possibilities in case of faults at the different layers seen before.

The Figure 3.1.2 shows the representation of the previous example using a standardized graphical notation for drawing business process (BPMN 1.1, [25]).
3. Motivating Example

Figure 3.1.2. Travel Agency BPEL representation (using BPMN 1.1 [25])

3.1 Error Sending IP Datagrams

In this situation, everything is all right, the travel agency did the request for reservation but when the hotel sends the confirmation, some problems in the network occur.

An IP datagram cannot reach its destination in different situations:
- if the gateway does not have the buffering capacity to forward a datagram,
- if the gateway cannot direct the host to send traffic,
- etc.

The intermediate host (usually a router) sends back to the initial host (travel agency) an ICMP message to inform about the concrete problem (“Destination Unreachable”, “Time Exceeded”, “Source Quench Message”, “Parameter Problem Message”…). ICMP error messages [22].

This ICMP error message is passed up from the IP layer to the TCP layer, which depending of the kind of ICMP error will act consequently. See section 4.1.2.3.

This fault is propagated through the Web Service Stack until reach the BPEL process which returns a generic fault and proceeds to stops. Therefore, the customer receives no confirmation by the travel agent.
### 3.1.1 Destination Unreachable Example

In case of no route is found to the destination host (Figure 3.1.3), the host or its inbound gateway send back an ICMP message type 3, Destination Unreachable, code 0, Network unreachable error. Informing that the IP module cannot deliver the datagram because is not able to find a route to the destination.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>message</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Network unreachable error.</td>
<td>“No route to host”</td>
</tr>
</tbody>
</table>

ICMP error message is passed up to the Transport layer. See section 2.1.2 Communication between layers. TCP must not abort the connection, and it should report the soft error condition directly to the application layer with an upcall to the ERROR_REPORT routine.
3. Motivating Example

3.2 Error Sending TCP Segments

Everything is all right, the travel agency did the reservation request successful, but when the hotel sent back the confirmation, it had some problems in the transport, due to one or more segments were lost.

TCP ([20]) is a connection-oriented, end-to-end reliable protocol. TCP entity carries out all the functions of error and flow control on the later stage of data transfer.

If some ICMP error message is reported from the IP layer, TCP acts in relation to this ICMP code. See section 4.1.2.3. ICMP error messages [22].

In case of a transfer (by the travel agency), in which a segment is not confirmed by the receiver TCP entity (hotel), this segment or segments will be retransmitted again after a period of time (timeout). The same action is done when a segment is lost or takes more time than expected to reach its destination.

3.2.1 TCP Segment Lost

In case of a TCP segment lost, TCP, as a reliable protocol, use a technique called retransmission to solve this situation. When TCP data is sent, a timer is started. If the timer expires before an acknowledgement arrives, TCP retransmits the data. This is illustrated in Figure 3.2.1.

![Figure 3.2.1: Example retransmission](image)

- 24 -
The host on the left is sending data, and the host on the right is receiving data. TCP must be ready to retransmit any packet whatever this is.

The number of retries has been predefined before with the aim of do not have an infinite number of retries. Excessive retransmissions indicate a possible route/gate failure, end node failure or possible mis-configuration.

In case that the number of configured retransmission retries is exceeded, the connection will be abort. In that case, and like was described in the previous section, de BPEL process will stops and the customer will not receive a confirmation.

3.3 Error in Application Layer

This time the travel agency try to do the reservation, but the hotel web server does not work or is not able to attend the travel agency petition.

In this case the TCP layer does not know about the existing problem, due to is not a transport problem; all the data that was buffering to it from the upper layer was delivery successful to the receiver TCP host (whose sent up to the upper layer). Therefore is the application layer the responsible to send back the "server error" message by HTTP (5xx).

Other way to arise an HTTP 5xx error is that the number of TCP retries exceeds the number predefined which is interpreted like the server is not able to attend the current petition at this moment.

If the situation is that the HTTP client of the travel agency has the problem, the situation is similar, but in this case it is a “client error” (4xx).

These faults are propagated through the Web Service Stack until reach the BPEL process which returns a generic fault and proceeds to stops.
3. Motivating Example

3.3.1 HTTP Service Unavailable

In this situation, the server is unable to handle the request due to a temporary overloading or maintenance of the server. Therefore, the server responds with an HTTP response message which status code is 503, meaning that the service is unreachable at this moment. This Response will be send passed down through the server stack and bottom-up through the client stack as shown in section 2.1.2. (See Figure 3.3.1) This is a temporary condition which will be alleviated after some delay. If the delay in known, it may be indicated in a Retry-After header. If not, the client should handle the response as it would for a 500 response.

In the upper layer and like was show before, the BPEL process will stops and the customer will not receive a response.

Figure 3.3.1. Http Server Unavailable.
3.4 Error in Messaging - SOAP

For this last case, imagine that the message SOAP (binding in HTTP) sent by the travel agency, is not correct (inexistent procedure, missing parameters ...). In this case, when the Web Service (hotel) discovers the error, it sends back a SOAP/HTTP message with the corresponding error to the client, to inform him that has been the error cause.

One important think is to underlying that HTTP is used at protocol and SOAP only as encoding (SOAP binding in HTTP). For this reason the HTTP protocol has to be informed about the error. The server must issue an HTTP 500 "Internal Server Error" response and include a SOAP message in the response containing a SOAP Fault element indicating the SOAP processing error.

Figure 3.3.2 shows the logical state transitions at the requesting SOAP node (travel agency) during the lifetime of the message exchange.

![Requesting SOAP node state transitions](image)

Table 3.4.1 shows the transitions that take place when a requesting SOAP node receives an HTTP error. For some status codes there is a choice of possible next state.

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Reason phrase</th>
<th>NextState</th>
</tr>
</thead>
<tbody>
<tr>
<td>4xx</td>
<td>Client Error</td>
<td></td>
</tr>
<tr>
<td>400</td>
<td>Bad Request</td>
<td>&quot;Sending+Receiving&quot;, &quot;Receiving&quot; or &quot;Fail&quot;</td>
</tr>
<tr>
<td>401</td>
<td>Unauthorized</td>
<td>&quot;Requesting&quot; or &quot;Fail&quot;</td>
</tr>
<tr>
<td>405</td>
<td>Method not allowed</td>
<td>&quot;Fail&quot;</td>
</tr>
<tr>
<td>415</td>
<td>Unsupported Media Type</td>
<td>&quot;Fail&quot;</td>
</tr>
<tr>
<td>5xx</td>
<td>Server Error</td>
<td></td>
</tr>
<tr>
<td>500</td>
<td>Internal Server Error</td>
<td>&quot;Sending+Receiving&quot;, &quot;Receiving&quot; or &quot;Fail&quot;</td>
</tr>
</tbody>
</table>

Table 3.4.1: Requesting SOAP node transitions
In cases where "Fail" is one of the choices, the transition is dependent on whether a SOAP message is present in the HTTP response. If a SOAP message is present, the next state is "Sending+Receiving" or "Receiving", otherwise the next state is "Fail".

When the Requesting SOAP node knows that the situation was unsuccessful (HTTP code 500), it will be transits to a fail state, like was shown in Table 3.4.1. At this moment, the SOAP node can show that an error occurred during the transaction: "Message send failed".

Table 3.4.2 shows a sample SOAP/HTTP message response indicating a failure handling the SOAP body.

<table>
<thead>
<tr>
<th>HTTP/1.1 500 Internal Server Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Content-Type: application/soap+xml; charset=&quot;utf-8&quot;</td>
</tr>
<tr>
<td>Content-Length: nnnn</td>
</tr>
</tbody>
</table>

```xml
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  <env:Body>
    <env:Fault>
      <env:Code>
        <env:Value>env:Sender</env:Value>
        <env:Subcode>
          <env:Value>rpc:BadArguments</env:Value>
        </env:Subcode>
      </env:Code>
      <env:Reason>
      </env:Reason>
      <env:Detail>
        <e:myFaultDetails xmlns:e="http://travelagency.example.de/faults" >
          <e:message> Missing parameters for booking </e:message>
        </e:myFaultDetails>
      </env:Detail>
    </env:Fault>
  </env:Body>
</env:Envelope>
```

Table 3.4.2. SOAP/HTTP error
3.4.1 SOAP mustUnderstand

A basic delivery of a single SOAP message needs two participants, SOAP sender and SOAP receiver. Each SOAP message is sent by SOAP sender and received by SOAP receiver.

A request containing some booking dates is sent by a SOAP sender to a SOAP receiver where some application is invoked (BPEL process). The application processes the request and generates a response, which is returned to the SOAP sender that originated the request. The SOAP sender is informed of the status (successful or otherwise) of the request message delivery.

In our situation, an immediate child element of the SOAP Header element, with the mustUnderstand attribute set to "1", was not understood. Then, the response returned to the SOAP sender, has the fault code “mustUnderstand” in its body (Table 3.4.3).

```
<env:Body>
  <env:Fault>
    <env:Code>
      <env:Value>env:mustUnderstand</env:Value>
    </env:Code>
    <env:Reason>
      <env:Text xml:lang="en">Must Understand</env:Text>
    </env:Reason>
  </env:Fault>
</env:Body>
```

Table 3.4.3. SOAP fault mustUnderstand.
3. Motivating Example

Figure 3.4.1. SOAP/HTTP Request/Response [37]

If the underlying transport protocol is HTTP, the SOAP request is sending as a payload in a synchronous HTTP POST request message. Following, the receipt of the request, process the received information and return a response as the payload of a HTTP response message with appropriate status codes. In this case, the Server must issue an HTTP 500 "Internal Server Error" response and include a SOAP message in the response containing a SOAP Fault “mustUnderstand”.
4. Fault Handling in the Different Layers

In this chapter will be described in more detail the different layers that have previously been identified in chapter 2. Moreover, the different layer's mechanisms to deal with errors and failures that occur in them and how these faults are reflected (ICMP messages, error Codes...) will be described.

Along the explanation of this chapter, the beginning of each section will have a figure that highlight the current layer.

4.1 Fault-Handling in TCP/IP

![Web Services stack. Transport Layer (TCP/IP).](image)

4.1.1 Fault Handling in IP

The Internet Protocol (IP for short, [18]) is a network layer protocol (Figure 4.1.1). IP provides the functions necessary to deliver an Internet datagram (a package of bits) from a source to a destination over an interconnected system of networks.
The Internet Protocol does not have control error or control flow mechanisms, is for this that such protocol is not totally reliable. IP Errors detected may be reported via the Internet Control Message Protocol (ICMP, [19]) which is implemented in the Internet protocol module.

**4.1.1.1 ICMP Protocol**

Internet Control Message Protocol (ICMP for short, [19]) is part of the Internet Protocol suite [18]. ICMP messages are usually generated in response to errors in IP datagrams (as specified in RFC-1122) or for diagnostic or routing purposes.

They are several situations in which an ICMP message is sent:
- when a datagram cannot reach its destination,
- when the gateway does not have the buffering capacity to forward a datagram,
- when the gateway can direct the host to send traffic on a shorter route
- others

**Brief list of Control Messages**

Described below are the most relevant control error messages. The full list can be found at [22].

**Destination Unreachable**

ICMP message type 3 is generated by the host or its inbound gateway. This message informs that the IP module cannot deliver the datagram because the indicated protocol module or process port is not active. (See Table 4.1.1)

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>message</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Network unreachable error.</td>
<td>&quot;No route to host&quot;</td>
</tr>
<tr>
<td>1</td>
<td>Host unreachable error.</td>
<td>&quot;No route to host&quot;</td>
</tr>
<tr>
<td>2</td>
<td>Protocol unreachable error (the designated transport protocol is not supported).</td>
<td>&quot;Connection refused&quot;</td>
</tr>
<tr>
<td>3</td>
<td>Port unreachable error (the designated protocol is unable to inform the host of the incoming message).</td>
<td>&quot;Connection refused&quot;</td>
</tr>
<tr>
<td>4</td>
<td>The datagram is too big. Packet fragmentation is required but the 'don't fragment' (DF) flag is on.</td>
<td>&quot;Message too long&quot;</td>
</tr>
<tr>
<td>5</td>
<td>Source route failed error.</td>
<td>&quot;No route to host&quot;</td>
</tr>
<tr>
<td>6</td>
<td>Destination network unknown error.</td>
<td>&quot;No route to host&quot;</td>
</tr>
<tr>
<td>7</td>
<td>Destination host unknown error.</td>
<td>&quot;No route to host&quot;</td>
</tr>
</tbody>
</table>
Fault Handling Across the Web Services Stack

<table>
<thead>
<tr>
<th></th>
<th>Source host isolated error (obsolete).</th>
<th>&quot;No route to host&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>The destination network is administratively prohibited.</td>
<td>&quot;No route to host&quot;</td>
</tr>
<tr>
<td>10</td>
<td>The destination host is administratively prohibited.</td>
<td>&quot;No route to host&quot;</td>
</tr>
<tr>
<td>11</td>
<td>The network is unreachable for Type Of Service.</td>
<td>&quot;No route to host&quot;</td>
</tr>
<tr>
<td>12</td>
<td>The host is unreachable for Type Of Service.</td>
<td>&quot;No route to host&quot;</td>
</tr>
<tr>
<td>13</td>
<td>Communication administratively prohibited (administrative filtering prevents packet from being forwarded).</td>
<td>(ignored)</td>
</tr>
<tr>
<td>14</td>
<td>Host precedence violation (indicates the requested precedence is not permitted for the combination of host or network and port).</td>
<td>(ignored)</td>
</tr>
<tr>
<td>15</td>
<td>Precedence cutoff in effect (precedence of datagram is below the level set by the network administrators).</td>
<td>(ignored)</td>
</tr>
</tbody>
</table>

Table 4.1.1. ICMP type 3 messages [22]

**Time Exceeded Message**

ICMP message type 11 is generated by a gateway. This message informs of a discarded datagram due to the time to live field reaching zero. Other situation in which this message can be sent is when the host fails to reassemble a fragmented datagram within its time limit (in this case is the own host which send the message). (See Table 4.1.2)

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Time-to-live exceeded in transit.</td>
</tr>
<tr>
<td>1</td>
<td>Fragment reassembly time exceeded.</td>
</tr>
</tbody>
</table>

Table 4.1.2. ICMP type 11 messages

**Source Quench Message (Congestion Control)**

ICMP message type 4, code 0. Request the sender to decrease the traffic rate of messages to a router or host. The source quench message is sent by the gateway or host when it approaches its capacity limit in state of waiting until its capacity is exceeded.
4. Fault Handling in the Different Layers

Redirect Message

ICMP message type 5. Informs to the host of an alternate route to send the datagrams (redirect its routing information). Only gateways should send the redirect states and not be sent by Internet hosts. [22]. (See Table 4.1.3).

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Redirect for Network Error.</td>
</tr>
<tr>
<td>1</td>
<td>Redirect for Host Error.</td>
</tr>
<tr>
<td>2</td>
<td>Redirect for Type of Service and Network Error.</td>
</tr>
<tr>
<td>3</td>
<td>Redirect for Type of Service and Host Error.</td>
</tr>
</tbody>
</table>

Table 4.1.3. ICMP type 5 messages

Parameter Problem Message

ICMP message type 12, code 0. Informs about problems with the header parameters, such that it cannot complete processing the datagram. An incorrect argument in an option is a possibly source of such a problem.

4.1.2 Fault Handling in TCP

The Transmission Control Protocol (TCP for short, [20]) is a connection-oriented, end-to-end reliable protocol. The TCP provides for reliable communication between pairs of processes in host computers attached to communication networks.

The TCP meaning of reliable is that the protocol itself checks if everything that was transmitted was delivered at the receiving end. TCP allows retransmission of lost packets, thereby making sure that all data transmitted is received.

The TCP is situated into layered protocol architecture just above Internet Protocol (IP for short, [18]) which provides a way for the TCP to send and receive variable-length segments of information encapsulated in Internet datagram.

TCP provides, to the corresponding application process, a flow of bytes (byte-stream), connection-oriented and therefore reliable, and without congestion. It also provides a multiplexing and duplex (in both directions) service. These features TCP are breakdown below.
4.1.2.1 TCP Features

**Byte-stream**

The emitter TCP entity allows the application process to transmit a steady stream of bytes that the emitter TCP entity is collecting, numbering, and grouping in data units called segments. Therefore, the application process does not need to take care to delimit the data that it wants to transmit. In this context, the entity computes a TCP MSS (Maximum Segment) so that the IP datagram are consistent with the MTU (Maximum Transfer Unit) of the access network. This is one of the main differences over the UDP protocol, which we shall send blocks of a fixed maximum length and well-defined.

**Connection-Oriented**

Through the connection between the corresponding pair of sockets (client-server), the emitter TCP entity is in agreement with the receiving TCP entity to carry out all the functions of error and flow control on the later stage of data transfer.

**Errors Control**

These errors cover both physical and logical errors.

**Logical errors** (bytes of lost data, disordered, or duplicate). Are controlled through the use of the following mechanisms:

- **Timers**: are the waiting terms for the relevant confirmations of data bytes transmitted. Each data field bytes have associated his relevant timer and his expiry, that is, if the timer expires before they confirm any byte segment; the segment will be sent again. TCP protocol uses an adaptive algorithm to establish the value of the timer. This technique is based on measuring of the Round Trip Time (RTT).
- **Sequence Numbers**: are assigned to each byte transmitted. Each byte has its own sequence number.
- **Confirmations**: they allow the emitter TCP entity turn his transmission window to continue sending data.

**Physical errors** (reversed or changed bits). Consisting of two processes:

- **Detection**: using a checksum mechanism that applies to the entire segment.
- **Correction**: through broadcasts by the expiration of the corresponding timers. The receiving TCP entity eliminates the affected segment and does not send a retransmission request. On expiry of the timer, the emitter TCP entity transmits again the affected bytes.
Flow Control

This service prevents that an emitter TCP entity "B", transmit faster than other receiving TCP entity "A" is able to storing and processing. It is based on a sliding reception window mechanism, which defines a credit system for bytes transmission. This credit system indicates the number of bytes that the entity "A" can receive and handle.

Multiplexing

TCP offers a multiplexing service. The multiplexing refers to a TCP entity can give service to several application processes simultaneously or in parallel.

Full-duplex

The flow of information between TCP entities is simultaneous and bidirectional.

4.1.2.2 Congestion Control

Internet Congestion

Congestion is produced when one or more IP datagram are lost. Possible Internet congestion causes are: routers (unable to attempt more petitions), low capacity links, input traffic excess in one or more Internet networks (input rates exceed the capabilities of departure), etc.

If one of the links of the route is affected by congestion is enough to limit all the traffic, and consequently, the performance of the communication between end systems. This can block or stop working the applications.

4.1.2.3 ICMP Messages

TCP must act on an ICMP error message passed up from the IP layer, directing it to the connection that created the error.
**Destination Unreachable**

**Codes 0, 1 and 5**

TCP must not abort the connection, and also it should make the information available to the application. [22].

These errors conditions can be reported by TCP directly to the application layer with an upcall to the ERROR_REPORT routine. TCP can just note the message and report it to the application only when and if the TCP connection times out.

**Codes 2, 3 and 4**

TCP should abort the connection, hence to these are hard error conditions. [22].

**Time Exceeded Message**

This should be handled the same way as “Destination Unreachable” Codes 0, 1, 5 (see above).

**Source Quench Message**

TCP must react by slowing transmission on the connection is a “Source Quench” is raised. Is suggested procedure in this situation is to trigger a "slow start", as if a retransmission timeout had occurred.

**Parameter Problem Message**

This should be handled the same way as “Destination Unreachable” Codes 0, 1, 5 (see above).

**4.1.2.4 TCP Interfaces**

The TCP has two interfaces. One side to user or application processes, and on the other side, to a lower level protocol (such as Internet Protocol).
The interface between TCP and lower level protocol provides calls to send and receive datagrams. It is assumed there is a mechanism whereby the two levels can pass information to each other asynchronously.

The interface between an application process and the TCP consists of a set of calls. These calls allow to OPEN or CLOSE a connection, to SEND or RECEIVE data, or to obtain STATUS about a connection. It is also expected that the TCP can asynchronously communicate with application programs (e.g. reporting soft TCP error conditions to the application).

**TCP Application Layer Interface**

TCP must communicate with applications in the upper layer and a network system in the layer below. There are several messages defined for the upper-layer protocol to TCP communications.

The TCP to upper-layer protocol communication method is well-defined, consisting of a set of calls (primitives). (See Table 4.1.4).

<table>
<thead>
<tr>
<th>Command</th>
<th>Parameters Expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABORT</td>
<td>Local connection name.</td>
</tr>
<tr>
<td>ACTIVE-OPEN</td>
<td>Local port, remote socket.</td>
</tr>
<tr>
<td></td>
<td>Optional: ULP timeout, timeout action, precedence, security, options</td>
</tr>
<tr>
<td>ACTIVE-OPEN-WITH-DATA</td>
<td>Source port, destination socket, data, data length, push flag, urgent flag</td>
</tr>
<tr>
<td></td>
<td>Optional: ULP timeout, timeout action, precedence, security</td>
</tr>
<tr>
<td>CLOSE</td>
<td>Local connection name</td>
</tr>
<tr>
<td>FULL-PASSIVE-OPEN</td>
<td>Local port, destination socket</td>
</tr>
<tr>
<td></td>
<td>Optional: ULP timeout, timeout action, precedence, security, options</td>
</tr>
<tr>
<td>RECEIVE</td>
<td>Local connection name, buffer address, byte count, push flag, urgent flag</td>
</tr>
<tr>
<td>SEND</td>
<td>Local connection name, buffer address, data length, push flag, urgent flag</td>
</tr>
<tr>
<td></td>
<td>Optional: ULP timeout, timeout action</td>
</tr>
<tr>
<td>STATUS</td>
<td>Local connection name</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>UNSPECIFIED-PASSIVE-OPEN</td>
<td>Local port</td>
</tr>
<tr>
<td>Optional: ULP timeout, timeout action, precedence, security, options</td>
<td></td>
</tr>
<tr>
<td>CLOSING</td>
<td>Local connection name</td>
</tr>
<tr>
<td>ERROR</td>
<td>Local connection name, error description</td>
</tr>
<tr>
<td>OPEN-FAILURE</td>
<td>Local connection name</td>
</tr>
<tr>
<td>OPEN-ID</td>
<td>Local connection name, remote socket, destination address</td>
</tr>
<tr>
<td>OPEN-SUCCESS</td>
<td>Local connection name</td>
</tr>
<tr>
<td>STATUS</td>
<td>Local connection name, source port, source address, remote socket, connection state, receive window, send window, amount waiting ACK, amount waiting receipt, urgent mode, precedence, security, timeout, timeout action</td>
</tr>
</tbody>
</table>

Table 4.1.4. TCP to Upper-layer primitives

The TCP must not only accept commands, but must also return information to the processes it serves. The latter consists of:

- General information about a connection (e.g., interrupts, remote close, binding of unspecified foreign socket).
- Replies to specific user commands indicating success or various types of failure.

### 4.1.2.5 Timeout Example

Figure 4.1.2 shows an unidirectional transmission of data from TCP entity "A" to a TCP entity "B". It is assumed that in such transmission errors occur. It shows only the most significant header control information of each segment information SEC and ACK (Number and Sequence Number Confirmation). The sketch of this figure can remain the same as Figure 3.2.1, due to in both a TCP retransmission occurs.

In this scenario, the TCP entity A transmits 900 bytes grouped into three segments to be confirmed. Thus, the first octet is identified with the number 3 (SEC = 3). The datagram that encapsulates the second segment is lost along the way.

The TCP entity "B" confirms the first 300 bytes after its arrival (ACK = 303). The arrival of the first confirmation (ACK = 303) to the TCP entity "A" does not produce a
retransmission (this only occurs after the expiration of the relevant timer), but a removal in the transmission buffer of the copy of the first 300 bytes.

When the TCP entity "B" receives the third information segment, sends again the previous confirmation (ACK = 303). Upon expiry of the timers (due to not receive confirmations for) of the two last information segments sent by "A", they are transmitted again.

![Figure 4.1.2. TCP data transfer with timeout errors](image)

Subsequently, "B" confirms all received bytes CONF = 903. This latest confirmation has grouped the last two segments (SEC = 303 and SEC= 603).
4.2 Fault Handling in HTTP

![Web Services stack](image)

Figure 4.2.1. Web Services stack. Transport Layer (HTTP). [26]

The Hypertext Transfer Protocol (HTTP for short, [23]) is a request/response standard protocol between a client and a server. It is located in the OSI application layer (Figure 4.2.1).

HTTP is a protocol for distributed, collaborative, hypermedia information systems, which original purpose was to provide a way to publish and retrieve hypertext pages over the Internet.

An HTTP client initiates a request. A Transmission Control Protocol (TCP) connection is established to a particular port on a host (port 80 by default). An HTTP server listening on that port waits for the request message from the client. Upon receiving the request, the server sends back a status line, such as "HTTP/1.1 200 OK", and a message of its own, the body of which is maybe the requested file, an error message, or other information.
4. Fault Handling in the Different Layers

4.2.1 HTTP Message

HTTP messages consist of requests from client to server and responses from server to client.

Both types of message consist of a start-line (Request-Line or Status-Line), zero or more headers, an empty line (CRLF) indicating the end of the headers, and perhaps a message-body.

<table>
<thead>
<tr>
<th>Request-Line</th>
<th>Status-Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>(message-header CRLF)*</td>
<td>CRLF</td>
</tr>
<tr>
<td>[ message-body ]</td>
<td></td>
</tr>
</tbody>
</table>

HTTP message-header contains: general-header, request-header, response-header, and entity-header fields. Each header field consists of a name followed by a colon (":") and the field value.

HTTP message-body is used to carry the body associated with the request or response.

Transfer-length of a message is the length of the message-body as it appears in the message.

4.2.2 HTTP Status Response

An HTTP/1.1 client should monitor the network connection when sending a message-body in case than an error status occurs while it is transmitting the request. The client should immediately cease transmitting the body if it observes an error status. The client must close the connection if the body was preceded by a content-length header.

The server responds with an HTTP response message after receiving and interpreting the request message.

The status-line is the first line of a response message, consisting of the protocol version followed by a numeric status code and its associated textual phrase.

The status-code element is a 3-digit integer result code of the attempt to understand and satisfy the request. The textual-phase is intended to give a short textual description of the status-code.

The first digit of the status-code defines the class of response. The last two digits do not have any categorization role. There are 5 values for the first digit:

- 1xx: Informational - Request received, continuing process.
2xx: Success - The action was successfully received, understood, and accepted.
3xx: Redirection - Further action must be taken in order to complete the request.
4xx: Client Error - The request contains bad syntax or cannot be fulfilled.
5xx: Server Error - The server failed to fulfil an apparently valid request.

Our interest is focused on the error codes: Client Error (4xx) and Server Error (5xx).

4.2.2.1 Client Error: 4xx

The 4xx status code is intended for cases in which the client seems to have erred. Error codes are listed below. [23].

"400": Bad Request
"401": Unauthorized
"402": Payment Required
"403": Forbidden
"404": Not Found
"405": Method Not Allowed
"406": Not Acceptable
"407": Proxy Authentication Required
"408": Request Time-out
"409": Conflict
"410": Gone
"411": Length Required
"412": Precondition Failed
"413": Request Entity Too Large
"414": Request-URI Too Large
"415": Unsupported Media Type
"416": Requested range not satisfiable
"417": Expectation Failed

4.2.2.2 Server Error: 5xx

The server has erred or is not capable of performing the request. Error codes are listed below. [23].

"500": Internal Server Error
"501": Not Implemented
"502": Bad Gateway
"503": Service Unavailable
"504": Gateway Time-out
"505": HTTP Version not supported
4. Fault Handling in the Different Layers

4.2.3 HTTP Example

HTTP communicates over TCP/IP. An HTTP client connects to an HTTP server using TCP. After establishing a connection, the client can send an HTTP request message to the server:

```
POST /item HTTP/1.1
Host: 189.123.345.239
Content-Type: text/plain
Content-Length: 200
```

The server then processes the request and sends an HTTP response back to the client. The response contains a status code that indicates the status of the request:

```
200 OK
Content-Type: text/plain
Content-Length: 200
```

In the example above, the server returned a status code of 200. This is the standard success code for HTTP.

If the server could not decode the request, it could have returned something like this:

```
400 Bad Request
Content-Type: text/plain
Content-Length: 0
```
4.3 Fault Handling in SOAP

The following sections will show several kinds of SOAP code sketch samples. The Simple Object Access Protocol (SOAP for short) [5, 8, 9, 10, 11], is not a protocol for accessing objects any more nowadays. (See Figure 4.3.1).

It is a XML based standard that consists of three parts:
- Envelope that defines a framework for describing what is in a message and how to process it.
- Encoding rules, for expressing instances of application-defined datatypes.
- RPC representation that defines a convention for representing remote procedure calls and responses.

SOAP provides a way to communicate between applications running on different operating systems, with different technologies and programming languages.
4. Fault Handling in the Different Layers

4.3.1 SOAP Syntax

A SOAP message is an XML document containing the following elements:
- A required envelope element, which identifies the XML document as a SOAP message.
- An optional header element, which contains header information.
- A required body element that contains call and response information.
- An optional fault element, which provides information about errors arise while processing the message.

All the elements above are declared in the default namespace for the SOAP envelope, and the default namespace for SOAP encoding and data types.

4.3.1.1 SOAP Envelope Element

The root element of a SOAP message is the mandatory SOAP Envelope element. SOAP Envelope defines the XML document as a SOAP message.

4.3.1.2 SOAP Header Element

SOAP Header element is an optional element which contains application specific information about the SOAP message (like payment, authentication, etc). The Header element must be the first child element of the Envelope element in case that it is present.

4.3.1.3 SOAP Body Element

The mandatory SOAP Body element contains the actual SOAP message intended for the ultimate endpoint of the message.

4.3.1.4 SOAP Fault Element

If an error was arise during the SOAP treatment the error message is carried inside a Fault element.

In case that a Fault element is present, it must appear as a child element of the Body element. In SOAP message can only appear one Fault element.
The Fault element has two mandatory sub elements, code and reason, and possibly optional detail, node and role elements. (See Table 4.3.1).

<table>
<thead>
<tr>
<th>Sub Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="">env:Code</a></td>
<td>Holds one or two child elements, env:Value and env:Subcode. env:Value contains a value indicating the kind of error that occurred: (see bellow)</td>
</tr>
<tr>
<td>&lt;env: Reason&gt;</td>
<td>A human readable explanation of the fault.</td>
</tr>
<tr>
<td><a href="">env:Detail</a></td>
<td>Holds application specific error information related to the Body element.</td>
</tr>
<tr>
<td><a href="">env:Node</a></td>
<td>Provide information about which SOAP node on the SOAP message path caused the fault to happen.</td>
</tr>
<tr>
<td><a href="">env:Role</a></td>
<td>Identifies the role node that was operating at the point that the fault occurred.</td>
</tr>
</tbody>
</table>

Table 4.3.1 SOAP Fault elements

**SOAP Fault Codes:**

A similar mechanism to the 1xx, 2xx, 3xx etc status classes defined in HTTP is used in for the SOAP Fault Codes. The only difference is that instead of integers, they are defined as XML qualified names.

Faultcode values are separated by the character "." (dot) indicating that the left part of the dot is a more generic fault code value than the value to the right.

The values of the Value child element of the Code element are defined in Table 4.3.2:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VersionMismatch</td>
<td>Found an invalid namespace for the SOAP Envelope element.</td>
</tr>
<tr>
<td>MustUnderstand</td>
<td>An immediate child element of the Header element, with the mustUnderstand attribute set to &quot;1&quot;, was not understood.</td>
</tr>
<tr>
<td>DataEncodingUnknown</td>
<td>A SOAP header block or SOAP body child element targeted at the faulting SOAP node is scoped with a data encoding that the faulting node does not support.</td>
</tr>
</tbody>
</table>
Client | The message was incorrectly formed or contained incorrect information.
---|---
Server | There was a problem with the server so the message could not proceed.

Table 4.3.2. SOAP Fault codes

Additional fault subcodes may be created for use by applications or features. Such subcodes are carried in the Value child element of the Subcode element.

### 4.3.2 SOAP Processing Model

SOAP provides a distributed processing model that assumes a SOAP message originates at an initial SOAP sender and is sent to an ultimate SOAP receiver via zero or more SOAP intermediaries.

The description of how a SOAP receiver processes a SOAP message is made by SOAP processing model. It applies to a single message only. The SOAP processing model itself does not maintain any state or perform any correlation or coordination between messages.

#### 4.3.2.1 SOAP Roles and Nodes

A SOAP node, which is identified by a URI, can be:
- the initial SOAP sender,
- an ultimate SOAP receiver or
- a SOAP intermediary.

In processing a SOAP message, a SOAP node is can act in more SOAP roles. Along the processing of an individual SOAP message the roles assumed by a node must be invariant.

The default SOAP roles are defined in Table 4.3.3.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>next</td>
<td>Each SOAP intermediary and the ultimate SOAP receiver must act in this role.</td>
</tr>
</tbody>
</table>
Table 4.3.3. SOAP predefined roles

<table>
<thead>
<tr>
<th>Role</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>SOAP nodes must not act in this role.</td>
</tr>
<tr>
<td>ultimateReceiver</td>
<td>The ultimate receiver must act in this role.</td>
</tr>
</tbody>
</table>

In addition to these SOAP roles, other role names may be used as necessary to meet the needs of SOAP applications.

### 4.3.2.2 SOAP Header Blocks

A SOAP header block may carry a “mustUnderstand” attribute. When the value of such attribute is "true", the SOAP header block is said to be mandatory.

Mandatory SOAP header blocks apparently modify the semantics of other SOAP header blocks or SOAP body elements. Consequently, for every mandatory SOAP header block, that node have to process the header block or not process the SOAP message at all, and instead generate a fault.

### 4.3.2.3 SOAP Bodies

An ultimate SOAP receiver has to correctly process the immediate children of the SOAP body.

### 4.3.2.4 SOAP Processing Messages

This section describes a set of rules by which SOAP messages are processed:

- Determine the set of roles in which the node is to act.
- Identify all header blocks that are mandatory.
- If one or more of the mandatory SOAP header blocks are not understood by the node, then generate a single SOAP fault ("env:MustUnderstand"). If such a fault is generated, any further processing must not be done. SOAP body faults must not be generated in this step.
- Process all mandatory SOAP header blocks. A SOAP node may also choose to process non-mandatory SOAP header blocks.
- In the case of SOAP intermediary, and where the SOAP message require that the SOAP message to be sent along the SOAP message path, relay the message.
Failure is indicated by the generation of a fault. A message may contain or result in multiple errors during processing. SOAP node could reflect any single fault from the set of these possible faults.

4.3.2.5 Relaying Messages

SOAP message begins at an initial SOAP sender and is sent to an ultimate SOAP receiver via zero or more SOAP intermediaries. Relaying messages describes how message forwarding interacts with the SOAP distributed processing model.

SOAP defines two different types of intermediaries:
- forwarding intermediaries.
- active intermediaries.

**SOAP Forwarding Intermediaries**

One or more SOAP header blocks in a SOAP message, need that the SOAP message be forwarded to another SOAP node on behalf of the initiator of the inbound SOAP message. In this case, the processing SOAP node acts in the role of a SOAP forwarding intermediary.

Forwarding SOAP intermediaries have to process the message according to the SOAP processing model. In addition, they must:
- Remove all processed SOAP header blocks.
- Remove all non-relayable SOAP header blocks that were targeted at the forwarding node but ignored during processing.
- Retain all relayable SOAP header blocks that were targeted at the forwarding node but ignored during processing.

**SOAP Active Intermediaries**

In addition to the processing by forwarding SOAP intermediaries, active SOAP intermediaries assume additional processing that can modify the outbound SOAP. The results of such active processing could collision the interpretation of SOAP messages by downstream SOAP nodes. For example, an active SOAP intermediary might have removed and encrypted some or the entire SOAP header blocks found in the inbound SOAP message.
4.3.3 SOAP Protocol Binding

SOAP enables exchange of SOAP messages using a lot of underlying protocols. The formal set of rules for carrying a SOAP message within or on top of another protocol with the aim of exchange, is called a binding. SOAP binding is an integral part of SOAP.

The SOAP Protocol Binding Framework defines and describes:
- rules for the specification of protocol bindings,
- the relationship between bindings and SOAP nodes.

The definition and the format of SOAP parameters, and other information are defined in the WSDL document. The SOAP binding also extends WSDL with the extension elements of shows on Table 4.3.4. (See also section 4.4.1).

```
<definitions .... >
  <binding .... >
    <soap:binding style="rpc|document" transport="uri">
      <operation .... >
        <soap:operation soapAction="uri"? style="rpc|document"?>
          <input>
            <soap:body parts="nmtokens"? use="literal|encoded" encodingStyle="uri-list"? namespace="uri"?>
              <soap:header message="qname" part="nmtoken" use="literal|encoded" encodingStyle="uri-list"? namespace="uri"?/>
              <soap:headerfault message="qname" part="nmtoken" use="literal|encoded" encodingStyle="uri-list"? namespace="uri"?/>
            </soap:body>
          </input>
          <output>
            [same sketch than before]
          </output>
          <fault>*
            <soap:fault name="nmtoken" use="literal|encoded" encodingStyle="uri-list"? namespace="uri"?/>
          </fault>
        </operation>
      </binding>
      <port .... >
        <soap:address location="uri"/>
      </port>
    </soap:binding>
  </definitions>
```

Table 4.3.4 SOAP Binding extends WSDL [13]
4. Fault Handling in the Different Layers

4.3.3.1 SOAP in HTTP

Binding SOAP in HTTP provides the advantage of being able to use the decentralized flexibility of SOAP with the rich feature set of HTTP.

SOAP obviously follows the HTTP request/response message model providing SOAP request parameters in a HTTP request and SOAP response parameters in a HTTP response.

When SOAP entity bodies are included in HTTP messages, HTTP applications must use the media type "text/xml" (according to RFC 2376).

The same semantics used in the HTTP Status codes for communicating status information in HTTP is followed in SOAP/HTTP binding.

If a SOAP error is arise while processing a request, the SOAP HTTP server must issue an HTTP 500 "Internal Server Error" response including a SOAP message in the response. This SOAP message has to contain a SOAP Fault element indicating the SOAP processing error.

4.3.4 SOAP Examples

The following tables will show several kinds of SOAP code samples.

SOAP Embedded in HTTP Request

<table>
<thead>
<tr>
<th>POST /Hotel HTTP/1.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host: <a href="http://www.hotelservice.com">www.hotelservice.com</a></td>
</tr>
<tr>
<td>Content-Type: text/xml; charset=&quot;utf-8&quot;</td>
</tr>
<tr>
<td>Content-Length: nnnn</td>
</tr>
<tr>
<td>SOAPAction: &quot;Some-URI&quot;</td>
</tr>
</tbody>
</table>

<env:Envelope
 xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
 env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
 <env:Body>
  <m:GetRoomPrice xmlns:m="Some-URI">
   <symbol>DIS</symbol>
  </m:GetRoomPrice>
 </env:Body>
</env:Envelope>

Table 4.3.5. SOAP embedded in HTTP Request
SOAP Embedded in HTTP Response

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<env:Envelope xmlns:env="http://www.w3.org/2001/06/soap-envelope" >
  ...
</env:Envelope>
```

Table 4.3.6. SOAP embedded in HTTP Response

SOAP Fault VersionMismatch

```
HTTP/1.1 500 Internal Server Error
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<?xml version="1.0" ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope"
  <env:Header>
    <env:Upgrade>
      <env:SupportedEnvelope qname="ns1:Envelope"
                              xmlns:ns1="http://www.w3.org/2003/05/soap-envelope"/>
      <env:SupportedEnvelope qname="ns2:Envelope"
                              xmlns:ns2="http://schemas.xmlsoap.org/soap/envelope"/>
    </env:Upgrade>
  </env:Header>
  <env:Body>
    <env:Fault>
      <env:Reason>
      </env:Reason>
    </env:Fault>
  </env:Body>
</env:Envelope>
```

Table 4.3.7. SOAP fault VersionMismatch
4.4 Fault Handling Specified in WSDL

WSDL faults occur due to synchronous operation invocations on partner Web Services. In WSDL, such faults are denoted with the “fault” element within the “operation” declaration. In WSDL faults are identified by the qualified name of the fault. [11, 13]. (See Figure 4.4.1).

WSDL does not require that unique fault names be used within the namespace used to define the operation. This implies that faults that have the same name and are defined within the same namespace will be considered as the same fault in BPEL.

4.4.1 SOAP Faults in WSDL

In this section will be show a typical WSDL that contains fault messages and how SOAP faults are defined in a WSDL file.
The following example (Table 4.4.1) of a WSDL file contains a fault message:

```
...<wsdl:portType name="HotelServicePT">
    <wsdl:operation name="reserveRoom">
        <wsdl:input message="tns:HotelServiceRequestMessage"/>
        <wsdl:output message="tns:HotelServiceResponseMessage"/>
        <wsdl:fault name="error" message="tns:HotelServiceFaultMessage"/>
    </wsdl:operation>
</wsdl:portType>
...```

Table 4.4.1. WSDL Fault definition

The definition of the message structure is described in the WSDL as follows:

```
...<xs:complexType name="HotelServiceExceptionType">
    <xs:sequence>
        <xs:element name="faultstring" type="xs:string"/>
        <xs:element name="detail" type="xs:string"/>
    </xs:sequence>
</xs:complexType>
...```

Table 4.4.2. WSDL Fault description

Table 4.4.3 shows how SOAP fault is binding in the WSDL file for the previous fault declared for “reserveRoom” operation. Table 4.4.4 shows the SOAP message when this condition is raised.

```
<wsdl:binding name="HotelServiceSoapBinding" type="tns:HotelServicePT">
    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="reserveRoom">
        <soap:operation soapAction="" style="rpc"/>
        <wsdl:input>
            <soap:body
                namespace="http://www.HotelService.wsdl"
                use="literal"/>
        </wsdl:input>
    </wsdl:operation>
</wsdl:binding>
```
4. Fault Handling in the Different Layers

---

```xml
<wsdl:output>
  <soap:body
    namespace="http://www.HotelService.wsdl"
    use="literal"/>
</wsdl:output>
<fault name="error">
  <soap:fault name="error" use="literal"/>
</fault>
</wsdl:operation>
</wsdl:binding>

Table 4.4.3. WSDL binding

```xml
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  [...]
  <env:Body>
    <env:Fault>
      <env:Code>
        <env:Value>env:Sender</env:Value>
        <env:Subcode>
          <env:Value>rpc:HotelServiceFaultMessage</env:Value>
        </env:Subcode>
        [...]
      </env:Code>
    </env:Fault>
  </env:Body>
  [...]
</env:Envelope>

Table 4.4.4. SOAP Fault from WSDL binding
```
4.5 Fault Handling in BPEL

Faults in BPEL (Figure 4.5.1) can arise in various situations: [11]

- When a BPEL process invokes a synchronous Web Services operation, the operation might return a WSDL fault message which results in a BPEL fault.
- A BPEL activity can explicitly signal (“throw”) a fault.
- A fault can be thrown automatically, for example, when a join failure has occurred.
- The BPEL server might find error conditions in the run-time environment, network communications, or any other such reason. BPEL defines several standard faults. These standard faults are depending of the BPEL engine that is used and do not solve all these situations.

There are two kinds of faults in a BPEL server:
- business faults (See section 4.5.1) and
- run time faults (See section 4.5.2).

Fault handlers describe how the BPEL process reacts when a fault occurs. Fault handling in BPEL provides the mechanism for catching several kinds of faults, propagating faults, and nesting fault-handling operations.

Figure 4.5.1. Web Services stack. Business Processes (BPEL) [26]
While a BPEL process is running, it can find faults from invoked services or faults originated from within the BPEL process. Faults that are due to service invocations are normally defined on the WSDL operation, whereas faults originated from within the process are BPEL-defined faults and are identified in certain error situations.

In BPEL is defined a construct to enable the handling of faults named "faultHandler" that contains "catch" or/and "catchAll" constructs:

- <catch> faultHandler construct defines certain criteria (the fault name) for catching a fault. The <catch> is executed when a fault is thrown and meets those criteria.
- <catchAll> faultHandler construct handles any other occurrence of a fault, which are not caught by a <catch> statement.

In BPEL is possible to isolate sets of related activities by placing them in a scope element. Using scope allows control a fault's effect and any consequent action or compensation.

BPEL provides the “faultHandler” construct at three levels:

- As associated with a BPEL process.
- As associated with a scope.
- As associated with an invoke activity.

Scopes not only isolate fault handling, but also provide a nesting capability; in consequence, fault handlers can form a nested hierarchy. If a nested "faultHandler" is not able to handle a particular fault, it delegates this fault to the enclosing "faultHandler”. The "throw" and "rethrow" constructs are defined to propagate a fault throughout the fault-handling chain.

### 4.5.1 Business Faults

Business faults can be arise when an explicit “throw” activity is executed or an <invoke> activity gets a fault as response. For example, when a hotel name is not found in the database. The Web Service operation cannot complete successfully, and the service returns a fault.

### 4.5.2 Run-time Faults

In case of problems within the BPEL process or the communication with the Web Services, run-time faults can be arise. An internal process error occurs, and a standard BPEL fault is thrown. For example when the communication fails during the execution of a BPEL <invoke> activity.
These faults are not user defined and will not appear in the WSDL for a process or service.

### 4.6. Layer’s Faults

There are several types of error conditions. A service returns a fault in response to a request it cannot process. A process may also raise a fault internally. In addition, processes may raise faults in order to terminate normal processing. See Figure 4.6.1.

Numerous faults can appear through the Web Service stack, beginning from Transport Layer (TCP/IP) and ending on Business Processes Layer (BPEL processes). The following sections group each fault within the corresponding layer.

#### 4.6.1 Transport

Here are grouped all the faults generated due to the network or to the transport. ICMP ([19]) messages are typically generated in response to errors in IP datagrams or for diagnostic or routing purposes. TCP must act on an ICMP error message passed up from the IP layer to the connection that created the error.
ICMP messages are sent in several situations:
- when a datagram cannot reach its destination,
- when the gateway does not have the buffering capacity to forward a datagram,
- when the gateway can direct the host to send traffic on a shorter route
- etc.

The most relevant control error messages are listed in Table 4.6.1.

<table>
<thead>
<tr>
<th>Message Type</th>
<th>Message Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Destination Unreachable</td>
<td>The IP module cannot deliver the datagram because the indicated protocol module or process port is not active.</td>
</tr>
<tr>
<td>Time Exceeded Message</td>
<td>Inform of a discarded datagram due to the time to live field reaching zero.</td>
</tr>
<tr>
<td>Source Quench Message</td>
<td>Request the sender to decrease the traffic rate of messages to a router or host.</td>
</tr>
<tr>
<td>Source Quench Message (Congestion Control)</td>
<td>Request the sender to decrease the traffic rate of messages to a router or host.</td>
</tr>
<tr>
<td>Redirect Message</td>
<td>Informs the host to redirect its routing information.</td>
</tr>
<tr>
<td>Parameter Problem Message</td>
<td>Problems with the header parameters, such that it cannot complete processing the datagram.</td>
</tr>
</tbody>
</table>

Table 4.6.1. Transport error messages
In this section, we are going to list the possible HTTP error messages (Figure 4.6.2) due to the client (4xx) or the server (5xx) seems to have errored (Table 4.6.2).

<table>
<thead>
<tr>
<th>Client</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;400&quot;: Bad Request</td>
<td>&quot;500&quot;: Internal Server Error</td>
</tr>
<tr>
<td>&quot;401&quot;: Unauthorized</td>
<td>&quot;501&quot;: Not Implemented</td>
</tr>
<tr>
<td>&quot;402&quot;: Payment Required</td>
<td>&quot;502&quot;: Bad Gateway</td>
</tr>
<tr>
<td>&quot;403&quot;: Forbidden</td>
<td>&quot;503&quot;: Service Unavailable</td>
</tr>
<tr>
<td>&quot;404&quot;: Not Found</td>
<td>&quot;504&quot;: Gateway Time-out</td>
</tr>
<tr>
<td>&quot;405&quot;: Method Not Allowed</td>
<td></td>
</tr>
<tr>
<td>&quot;406&quot;: Not Acceptable</td>
<td></td>
</tr>
<tr>
<td>&quot;407&quot;: Proxy Authentication Required</td>
<td></td>
</tr>
<tr>
<td>&quot;408&quot;: Request Time-out</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.6.2. Application (HTTP) error
4. Fault Handling in the Different Layers

<table>
<thead>
<tr>
<th>Code</th>
<th>Error Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;409&quot;</td>
<td>Conflict</td>
</tr>
<tr>
<td>&quot;410&quot;</td>
<td>Gone</td>
</tr>
<tr>
<td>&quot;411&quot;</td>
<td>Length Required</td>
</tr>
<tr>
<td>&quot;412&quot;</td>
<td>Precondition Failed</td>
</tr>
<tr>
<td>&quot;413&quot;</td>
<td>Request Entity Too Large</td>
</tr>
<tr>
<td>&quot;414&quot;</td>
<td>Request-URI Too Large</td>
</tr>
<tr>
<td>&quot;415&quot;</td>
<td>Unsupported Media Type</td>
</tr>
<tr>
<td>&quot;416&quot;</td>
<td>Requested range not satisfiable</td>
</tr>
<tr>
<td>&quot;417&quot;</td>
<td>Expectation Failed</td>
</tr>
<tr>
<td>&quot;505&quot;</td>
<td>HTTP Version not supported extension-code</td>
</tr>
</tbody>
</table>

Table 4.6.2 HTTP error messages

4.6.3 Messaging Layer

![Messaging Layer diagram]

Figure 4.6.3. Messaging Layer error
The general causes that could generate a SOAP fault (Figure 4.6.3) are listed in the Table 4.6.3:

<table>
<thead>
<tr>
<th>Description</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Found an invalid namespace for the SOAP Envelope element.</td>
<td></td>
</tr>
<tr>
<td>An immediate child element of the Header element, with the mustUnderstand attribute set to &quot;1&quot;, was not understood.</td>
<td></td>
</tr>
<tr>
<td>A SOAP header block or SOAP body child element targeted at the faulting SOAP node is scoped with a data encoding that the faulting node does not support.</td>
<td></td>
</tr>
<tr>
<td>The message was incorrectly formed or contained incorrect information.</td>
<td></td>
</tr>
<tr>
<td>There was a problem with the server so the message could not proceed.</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.6.3. SOAP Faults cond.

In this layer could also arise faults due to the Addressing:
- Faults generated by a bad or missing WS-Addressing header (Invalid Message Information Header, Message Information Header Required),
- Faults due to the Action header is present and well formed, but the URI content does not match any action at the destination (Action Not Supported),
- Faults produced because is not possible to find a route to delivery the message to the destination (Destination Unreachable),
- Faults produced when for some reason the endpoint is not currently available (Endpoint Unavailable).
4.6.4 Description and Discovery

![Diagram showing WSDL fault message]

WSDL faults (Figure 4.6.4) occur due to synchronous operation invocations on partner Web Services. WSDL faults are identified by the name of the fault and the target namespace of the corresponding port-type used in the operation declaration.

4.6.5 Business Processes Layer

As was presented in section 4.5, there are two kinds of faults in BPEL: business faults and run time faults. These faults in can arise in various situations:

- When a BPEL process invokes a synchronous Web Service operation, the operation might return a WSDL, fault message, with results in a BPEL fault.
- A BPEL process can explicitly signal (“throw”) a fault.
- A fault can be thrown automatically, for example, when a join failure has occurred (Figure 4.6.5).
- The BPEL server might find error conditions in the run-time environment, network communications (these are important for the thesis), or any other such reason. BPEL defines several standard faults.
Fault Handling Across the Web Services Stack

Figure 4.6.5. BPEL error message
5. Motivating Example Extended

As has been seen in chapter 3 (Motivating Example), if there is any error in the communication between the travel agency and hotel service, BPEL process can not complete successfully. The process can complete successfully only if the fault is handled within a scope. Fortunately, BPEL provides a mechanism of Fault Handling to deal against faults.

5.1 Adding BPEL Fault Handler

Fault handling in a BPEL process takes over when a fault occurs within a scope. The business process defines custom activities inside the fault handler which are used to recover from the fault.

Within a fault handler several <catch> activities can be specified, in each one, the faults that we would like to catch and handle have to be indicated. Within a fault handler, is usual to specify several <catch> activities for specific faults and <catchAll> activity to handle the rest of the faults (Table 5.1.1).

```
<faulthandlers>
  <catch faultName="qname">
    […]
  </catch>
  <catch faultVariable="var">
    […]
  </catch>
  <catch faultName=" qname" faultVariable="var">
    […]
  </catch>
  <catchAll>
    […]
  </catchAll>
</faulthandlers>
```

Table 5.1.1 BPEL fault handler sketch.

A fault can occur for different reasons (web service operation cannot complete successfully, internal process error, <throw> or <rethrow> activity throws a fault…). When a fault occurs, normal processing is terminated, and control is transferred to the corresponding fault handler of the process or scope.

Returning to the example, a faultHandler has been added. In case of problems with the communication with the hotel service a fault will be throwing and it will be catch by our
faultHandler. At this moment the BPEL process will use the activities defined in the faultHandler to try to recover from the fault. In our case, the travel agency will ask for a reservation room in other hotel service (Hotel Service 2). In case that a fault occurs again in this new attempt, the BPEL process will not be complete successfully.

The Figure 5.1.1 shows the result of adding the faultHandler to our example. The Figure 5.1.2 shows the representation using a standardized graphical notation for drawing business process (BPMN 1.1, [25]).

Figure 5.1.1. Travel Agency Example with faultHandler
Once the new situation of the motivating example has been explained, the most important part is to define the attributes (used to specify which fault to handle) within the <catch> activity. (Table 5.1.2).

```xml
[...]
<scope name="Reservation">
    <faultHandlers>
        <catch faultName="?????????">
            <sequence>
                [...]
                <invoke name="BookRoom_2" partnerLink="HotelService_2" portType="ns2:HotelServ" operation="reserveRoom"
                       inputVariable="BookRoom_2_reserveRoom_InputVariable"
                       outputVariable="BookRoom_2_reserveRoom_OutputVariable"/>
                [...]
            </sequence>
        </catch>
    </faultHandlers>
</sequence>
```

Figure 5.1.2. Travel Agency with faultHandler. BPEL representation ([25])
The interest of this thesis focuses on possible faults in the Web Services Stack, and particularly in faults during the communication: transport layer and messaging layer (timeouts, web server not available, MustUnderstand...). For this reason, the fault that have to be catch by the BPEL <catch> activity has to be throw when one of this kind of situations occurs.

In the following chapter (Current Situation), will be described how two of the most relevant BPEL engines deal with this kind of faults.
6. Current Situation

During the execution of a BPEL process, different kinds of errors may occur throughout Web Services Stack layers (Business Process, Quality of Service, Messaging, Transport...). Currently, if a communications fails, the BPEL process does not receive a detailed error depending on the cause, and the process is simply suspended or terminated.

In previous chapters the different faults/errors that may occur in each layers of the Web Service Stack, from TCP/IP to BPEL have been described. Also, for each layer, it has been shown how they identify such errors (their fault structures).

In the following sections are going to be described how two BPEL engines run and deal with the faults. These two engines are: “Oracle BPEL Process Manager” and “Apache ODE”. The first one is one of the most complete BPEL engines and the second one provides the advantage of go through the code due to is open source.

6.1 Oracle BPEL Process Manager

Running Web Services is a two step process: publishing services and orchestration. Publishing means enabling Web Services to be accessible (interface) through a binding (protocol). Orchestration means to unite and coordinate these services into a business process. However, implement, execute and manage the logic of orchestration is complex and requires a consistent architecture and infrastructure.

Oracle BPEL Process Manager provides exactly that infrastructure and is also completely scalable. Oracle BPEL Process Manager is a run-time environment that facilitates the modelling and development of business processes based on BPEL language.

At this moment, Oracle BPEL Process Manager is one of the most complete BPEL servers available (such the one by IBM).

6.1.1 Components

The Oracle BPEL Process Manager has four main components:

- BPEL Server
- BPEL Console
- BPEL Designer (JDeveloper)
- Database
6. Current Situation

6.1.1.1 BPEL Server

The BPEL Server is composed by:

- Core BPEL engine,
- WSDL bindings,
- Integration services.

The core BPEL engine is the run-time environment where BPEL processes are deployed and executed. In addition provides support for important technologies in the Web Services stack, such as WS-Addressing, WS-ReliableMessaging, WS-Security…

An important feature of BPEL engine is that it also provides support for version control. This enables development of several versions of a business process. In our example was useful to have two versions of the TravelAgency, one in which everything runs OK and other in which the faults happens.

The communication with a BPEL processes deployed on a server is responsibility of the WSDL binding framework. Oracle BPEL Process Manager enables connectivity using protocols other than SOAP, such as HTTP GET and POST, JMS (Java Message Service), email, and sockets.

6.1.1.2 BPEL Console

One of the most remarkable characteristics of Oracle BPEL Process Manager is the friendly BPEL Console which we can deploy, manage, administer, and debug BPEL processes easily. BPEL Console contains: Visual process flows, Audit trails, Debugging view of processes, Process history and Management of BPEL domains and their configuration. This web-based interface was very helpful for the realization of this thesis, allowing see the error flow and a good interface for debugging.

6.1.1.3 BPEL Designer (JDeveloper)

JDeveloper [27] is a BPEL Designer that enables the development of BPEL processes in a graphical environment. Thank to this, is not necessary to write BPEL code by hand. Instead of writing code, is possible drag and drop activities to the process.

JDeveloper BPEL Designer can deploy the developed processes directly to the BPEL Server, and also provides support for some advanced Oracle features, such as in our case, the possibility to fix in the <catch> activity the run-time fault “remoteFault” or “bindingFault”.

- 72 -
For more information about Oracle BPEL Process Manager please refer to [34].

### 6.1.2 Dealing with Faults

The Oracle BPEL allows fault handling management through the "faultHandler" construct. This thesis’s main focus are communication faults, this kind of faults are raised when the BPEL process run an <invoke> activity and some errors occurs during the communication with the invoked service (timeouts, HTTP 500 Internal Server Error, HTTP 400 Bad Request, remote service changes its interface, etc.).

Oracle BPEL Process Manager can deal with the two categories of faults in BPEL: business faults and runtime faults.

First ones, business faults, occur when a fault is the result of an <invoke> activity or when it is executed an explicit <throw> activity. In both cases business faults are specified by the BPEL process and are defined in the WSDL file.

The second ones, in contraposition to the business faults, do not appear defined in the WSDL file due to be not user defined. Oracle BPEL Process Manager defines 10 standard faults [35]:

- selectionFailure,
- conflictingReceive,
- conflictingRequest,
- mismatchedAssignmentFailure,
- joinFailure,
- forcedTermination,
- correlationViolation,
- uninitializedVariable,
- repeatedCompensation and
- invalidReply.

So far Oracle BPEL Process Manager deals with faults as any other BPEL engine, but there is a difference, Oracle also provides two more runtime faults [36]:

- bindingFault,
- remoteFault.

#### 6.1.2.1 BindingFault

These kinds of faults are raised when a problem with the binding happen. For example, in the case that a remote service has upgraded and the interface has changed.
Table 6.1.1 shows the possible fault codes:

<table>
<thead>
<tr>
<th>FaultCode</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>VersionMismatch</td>
<td>An invalid namespace for the SOAP Envelope element was found.</td>
</tr>
<tr>
<td>MustUnderstand</td>
<td>A SOAP mustUnderstand attribute with a value of &quot;1&quot; and the immediate child element of the SOAP Header element is not understood or not obeyed.</td>
</tr>
<tr>
<td>Client.GenericError</td>
<td>Generic error at client side</td>
</tr>
<tr>
<td>Client.WrongNumberOfInputParts</td>
<td>input message part number mismatch</td>
</tr>
<tr>
<td>Client.WrongNumberOfOutputParts</td>
<td>output message part number mismatch</td>
</tr>
<tr>
<td>Client.WrongTypeOfInputPart</td>
<td>input message part type error</td>
</tr>
<tr>
<td>Client.WrongTypeOfOutputPart</td>
<td>output message part type error</td>
</tr>
<tr>
<td>Server.GenericError</td>
<td>Generic error at server side</td>
</tr>
<tr>
<td>Server.NoService</td>
<td>Server is up but no service</td>
</tr>
<tr>
<td>Server.NoHTTPSOAPAction</td>
<td>Request is missing HTTP SOAP Action</td>
</tr>
<tr>
<td>Server.Unauthenticated</td>
<td>Request is not authenticated</td>
</tr>
<tr>
<td>Server.Unauthorized</td>
<td>Request is not authorized</td>
</tr>
</tbody>
</table>

Table 6.1.1 Oracle BindingFault faults codes.

6.1.2.2 RemoteFault

These kind of faults are raised when a problem with the communication with the Service invoked happen. For example, in case that a remote service has been called at TCP port 2020 and the Server is listening at TCP port 2120, in this case the fault code returned is “ConnectionRefused”.

Table 6.1.2 shows the possible fault codes:

<table>
<thead>
<tr>
<th>FaultCode</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConnectionRefused</td>
<td>The remote server is not up. (Calling at TCP port 2020 and the Server is listening at 2120).</td>
</tr>
<tr>
<td>WSDLReadingError</td>
<td>Fail to read WSDL</td>
</tr>
<tr>
<td>GenericRemoteFault</td>
<td>Generic remote fault</td>
</tr>
</tbody>
</table>

Table 6.1.2 Oracle RemoteFault faults codes.

In the next section (7.1) these type of runtime faults will be applied to the motivating example in order to solve the problem of a communication fault.


6.2 Apache ODE

Orchestration Director Engine [29] it is an orchestration service that implements the WS-BPEL specification. Apache ODE provides infrastructure to implement, execute and manage the logic of orchestration. The implementation will also support Message/Event to process correlation.

ODE is an Apache incubator project focused in ESB integration and, of course, integration with other Apache products, such as Axis2, Tomcat or Geronimo. This is an advantage due to ODE use some of these technologies as shown in the following lines.

ODE uses Axis2 for sending SOAP messages to remote Web Services (BPEL <invoke>). In the same way, Axis2 uses components of Apache Commons HTTPClient to transfer HTTP messages. Finally, Sun's Sockets are used by HTTPClient to perform the corresponding TCP calls.

6.2.1 Used Technologies

For better understanding, will be briefly described the components used by ODE:

- Apache Axis2
- Apache HTTPClient
- Apache Tomcat

6.2.1.1 Apache Axis2

Apache Axis2 [31] is a java based open source implementation that provides the basis for the web services stack. Axis2 supports SOAP (Simple Object Access Protocol) and REST (Representational State Transfer).

Axis2 has a modular and flexible architecture which allows an easier support of web services specifications. Axis2 architecture has more scalability than its predecessor Axis, to provide improved performance, and to support the changes better. The Axis2 core architecture reflects these dynamics and the engine provides flexibility to support different message-exchange patterns [32]:

- Send SOAP messages.
- Receive and process SOAP messages.
- Create a Web service out of a plain Java class.
- Create implementation classes for both the server and client using WSDL.
- Easily retrieve the WSDL for a service.
• Send and receive SOAP messages with attachments.
• Create or utilize a REST-based Web service.
• Create or utilize services that take advantage of the WS-Security, WS-ReliableMessaging, WS-Addressing, WS-Coordination, and WS-Atomic Transaction recommendations.
• Use Axis2's modular structure to easily add support for new recommendations as they emerge.
• Supports for multiple transports such as HTTP, JMS, SMTP, and TCP.

Axis2 supports synchronous and asynchronous interactions, providing flexibility at the transport layer by allowing responses asynchronously via the same or different transport (i.e.: request comes via HTTP and the processing is done asynchronously, the reply can be sent via JMS).

6.2.1.2 Apache HttpClient

The most used protocol on the Internet nowadays is the Hyper-Text Transfer Protocol (HTTP). Web services, network-enabled appliances and the growth of network computing continue to expand the role of the HTTP protocol beyond user-driven web browsers, while increasing the number of applications that require HTTP support.

Although the java.net package provides basic functionality for accessing resources via HTTP, it doesn't provide the full flexibility or functionality needed by many applications. The Jakarta Commons HttpClient component seeks to fill this void by providing an efficient, up-to-date, and feature-rich package implementing the client side of the most recent HTTP standards and recommendations.

Designed for extension while providing robust support for the base HTTP protocol, the HttpClient component may be of interest to anyone building HTTP-aware client applications such as web browsers, web service clients, or systems that leverage or extend the HTTP protocol for distributed communication. [30].

6.2.1.3 Apache Tomcat

A Service Container is where the services are executed, and their “Life Cycle” controlled. The service containers provide the means to deploy services in their execution containers. Apache ODE is integrated in Tomcat copying ode.war in the Tomcat webapps folder.

Apache Tomcat is the Servlet/JSP container that is used in the official Reference Implementation for the Java Servlet and JavaServer Pages technologies, specifications developed by SUN under the Java Development Process.
Nowadays is one of the most known, if not the most, servlet container used in all level of projects in the market.

Apache Tomcat is a well known project, widely reviewed, its architecture and its features, so for that purpose please refer to the references section. [28].

### 6.2.2 Dealing with Faults

In our Web Service Stack, ODE uses BPEL faults in its engine. Axis2 has acknowledged about SOAP faults and various exceptions that can happen in the transport layer. In addition, HTTP knows errors and exceptions at TCP level (such as logical errors and transport errors, socket errors...).

So far, the faults produced in the communication (i.e. transport layer) along with a call to a Web Service by a BPEL process (<invoke>) are replaced by a generic AxisFault. Thus, all information regarding such transport error is lost.

ODE as Oracle BPEL Process Manager can deal with the two categories of faults in BPEL: business faults and runtime faults.

Runtime faults do not appear defined in the WSDL description due to be not user defined. Apache ODE defines the same standard faults than Oracle BPEL Process Manager:

- **selectionFailure**: a selection operation performed in either a function, such as bpws:getVariableData, or an assignment encounters an error.
- **conflictingReceive**: more than one <receive> activity are enabled simultaneously for the same partner link, port type, operation, and correlation set(s).
- **conflictingRequest**: more than one synchronous inbound request is active on the same partner link for a particular port type, operation and correlation set(s).
- **mismatchedAssignmentFailure**: incompatible types are encountered in an assign activity.
- **joinFailure**: join condition of an activity evaluates to false.
- **forcedTermination**: a fault occurs in an enclosing scope.
- **correlationViolation**: the contents of messages processed in an <invoke>, <receive>, or <reply> activity do not match specified correlation information.
- **uninitializedVariable**: there is an attempt to access the value of an uninitialized part in a message variable.
- **repeatedCompensation**: an installed compensation handler is invoked more than once.
- **invalidReply**: a reply is sent on a partner link, port type, and operation for which the corresponding <receive> with the same correlation has not been carried out.
Apache ODE can throw also some specific faults relating to specific activities:

- ambiguousReceive: thrown when more than one inbound message activities are in a position to consume the same message. For example when two activities want to consume the same message (this can happen in the <receive> activity).
- invalidVariables: his fault is thrown if a variable contains data that is inconsistent with its schema (by the <assign> and <validate> activities).
- subLanguageExecutionFault: this fault is thrown by the <assign> activity when evaluating the expression, or the query generated an error.

As can be appreciated, Apache ODE does not offer a specific runtime faults as Oracle’s BPEL Process Manager does, therefore can not be treated any communication fault by the same way. The aim in the next chapter shall be proposing a new fault definition to be thrown when the communication with the remote service fails.
7. Proposed Solution

Returning to our example, our attention focuses on the faults that can happen in the communication between the "Travel Agency" and "Hotel Service 1". If any of these communications (highlighted in the Figure 7.1.1) fails, the faultHandler have to be activated (executed) and "run" the activities it contains. To achieve this purpose, the communication fault must be collected by the <catch> activity defined within the faultHandler.

![Figure 7.1.1 Communication Faults in the motivating example](image)

This chapter will propose a solution to this situation and will apply it to our motivating example.

7.1 Oracle BPEL Process Manager

As was described in the previous chapter (Current Situation), Oracle BPEL Process Manager defines two additional types of runtime faults more: "RemoteFault" and "BindingFault".
7. Proposed Solution

The runtime "RemoteFault" include faults generated when is not able to contact the server, either due to a malfunctioning or because it is not possible to receive a response from it. This can be translated or resembled to the faults produced by the underlined in Figure 7.1.1.

The runtime “BindingFaults” includes faults generated in the messaging layer such us VersionMismatch, MustUnderstand, Client and Server fault codes.

Therefore in Oracle BPEL Process Manager could be enough to put the "RemoteFault" and “BindingFault” into the <catch> activity within the faultHandler. As shown in Table 7.1.1.

```
<scope name="Reservation">
  <faultHandlers>
    <catch faultName="bpelx:remoteFault">
      <sequence>
        […]
        <invoke name="BookRoom_2" partnerLink="HotelService_2"
        portType="ns2:HotelServ" operation="reserveRoom"
        inputVariable="BookRoom_2_reserveRoom_InputVariable"
        outputVariable="BookRoom_2_reserveRoom_OutputVariable"/>
        […]
      </sequence>
    </catch>
    <catch faultName="bpelx:bindingFault">
      <sequence>
        […]
        <invoke name="BookRoom_2" partnerLink="HotelService_2"
        portType="ns2:HotelServ" operation="reserveRoom"
        inputVariable="BookRoom_2_reserveRoom_InputVariable"
        outputVariable="BookRoom_2_reserveRoom_OutputVariable"/>
        […]
      </sequence>
    </catch>
  </faultHandlers>
  <sequence>
    […]
    <invoke name="BookRoom_1" partnerLink="HotelService_1"
    portType="ns1:HotelService" operation="reserveRoom"
    inputVariable="BookRoom_1_reserveRoom_InputVariable"
    outputVariable="BookRoom_1_reserveRoom_OutputVariable"/>
    […]
  </sequence>
</scope>
```

Table 7.1.1. “RemoteFault” and “BindingFault” applied to our example.
7.1.1 Applied to the Example

In the case in which in our example there was no fault and using the Oracle BPEL Console, the next result would be achieved:

```xml
<TravelAgencyProcessResponse>
  Confirmation by Hotel 1
</TravelAgencyProcessResponse>
```

Figure 7.1.2 Example Flow without faults

In the previous situation of our example (without define "RemoteFault" and "BindingFault" within the <catch> activity), if a communication fault occurred between "Travel Agency" and "Hotel Service 1" the BPEL process would have been stopped or suspended after the call to <invoke> activity.
Figure 7.1.2 shows how the flow is when have been executing the previous situation. In Figure 7.1.3 are represented the flow in case that the <catch> activity is not defined. These two figures have been obtained using the Oracle BPEL Console.

Several types of faults in communication are captured by the "RemoteFault" and “BindingFault” Oracle BPEL runtime faults. In our case we have obtained the following outputs depending on the situation of error that has been produced.

**VersionMismatch**

This fault is produced when a SOAP node found an invalid element information item instead of the expected Envelope element information item. See Table 7.1.2.

```
"{http://schemas.oracle.com/bpel/extension}bindingFault"

-<messages>
  -<input>
    -<BookRoom_1_reserveRoom_InputVariable_1>
      […]
    </BookRoom_1_reserveRoom_InputVariable_1>
  </input>
```
Fault Handling Across the Web Services Stack

Table 7.1.2. Oracle VersionMismatch.

**Timeout**

This fault is produced when is introduced a high delay in the "Hotel Service 1" response, this delay is more than the expected for the "Travel Agency" causing the expiration of its timer and the associated "timeout". See Table 7.1.3.

"{http://schemas.oracle.com/bpel/extension}remoteFault"

```xml
<message xmlns="http://schemas.oracle.com/bpel/extension">
  <input>
    <BookRoom_1_reserveRoom_InputVariable>
      [...]
    </BookRoom_1_reserveRoom_InputVariable>
  </input>
  <fault>
    <remoteFault xmlns="http://schemas.oracle.com/bpel/extension">
      <summary>
        when invoking locally the endpoint \\
        "http://localhost:8080/axis/services/HotelService_1/1.0", ; \\
        nested exception is: \\
        com.oracle.bpel.client.delivery.ReceiveTimeOutException: \\
        Waiting for response has timed out…
      </summary>
    </remoteFault>
  </fault>
</message>
```
7. Proposed Solution

Server Down

In this case "Hotel Service 1" is not online, you can not access to it and therefore to the description of its services (HotelService.wsdl). This can happen due to the Tomcat Servlet is switched off or because there is not access to network connection that contains the service. See Table 7.1.4.

```
"{http://schemas.oracle.com/bpel/extension}remoteFault"

   -<messages>
       -<input>
           [...]  
       </input>
       -<fault>
           -<remoteFault xmlns="http://schemas.oracle.com/bpel/extension">  
               -<part name="code">  
                   <code>
                       WSDLReadingError
                   </code>
               </part>
               -<part name="summary">
                   <summary>
                       Fault reading wsdl.
                       An error was happened readind wsdl from:
                       "http://localhost:8080/axis/services/HotelService_1?wsdl",
                       caused by: java.net.ConnectException. : Connection refused: connect". Be sure that the wsdl exist on this URL direction and that is valid.
                   </summary>
               </part>
           </remoteFault>
       </fault>
   </messages>
```

Table 7.1.3. Oracle Timeout.

Table 7.1.4. Oracle WSDLReadingError.
Service not Available

In this situation "Hotel Service 1" is online at the time that "Travel Agency" do the request but immediately after the server crashes and can not respond to the request. See Table 7.1.5.

"{http://schemas.oracle.com/bpel/extension}remoteFault"

```xml
<messages>
  <input>
    <BookRoom_1_reserveRoom_InputVariable>
      [...] 
    </BookRoom_1_reserveRoom_InputVariable>
  </input>
  <fault>
    <remoteFault xmlns="http://schemas.oracle.com/bpel/extension">
      <part name="summary">
        <summary>
          exception on JaxRpc invoke: HTTP transport error: javax.xml.soap.SOAPException: java.security.PrivilegedActionException: javax.xml.soap.SOAPException: Bad response: 503 Service Unavailable at this moment
        </summary>
      </part>
    </remoteFault>
  </fault>
</messages>
```

Table 7.1.5. Oracle 503 Service Unavailable.

As can be seen in the previous tables, all the situations produce a "RemoteFault" or a "BindingFault". Therefore, the next step is to introduce the "RemoteFault" both into the <catch> activity with the aim of catch the fault and run the activities that are inside. Finally we can see the flow of this new situation making use of the Oracle BPEL Console again. (See Figure 7.1.4).

```
<TravelAgencyProcessResponse>
  Confirmation by Hotel 2 
</TravelAgencyProcessResponse>
```
Figure 7.1.4 Example Flow with “RemoteFault” and “BindingFault” defined within the <catch> activity
7.2 Apache ODE

In the case of Apache ODE are not defined standard fault that can be used to catch communication faults such as Oracle BPEL Process Manager.

In Apache ODE when a communication fault occurs during the call to a Web Service by a BPEL process (<invoke>), such faults are replaced by a generic AxisFault.

Taking both situations, the proposal to handle communication faults for this engine is to define a new fault in ODE which be thrown when a fault in the communication be arise, rather than throw a generic AxisFault. This new fault has to be catch as are catch the standard faults, being able to handle such as in Oracle (See Table 7.2.1).

```xml
<scope name="Reservation">
  <faultHandlers>
    <catch faultName="communicationFault">
      <sequence>
        […]
        <invoke name="BookRoom_2" partnerLink="HotelService_2"
          portType="ns2:HotelServ" operation="reserveRoom"
          inputVariable="BookRoom_2_reserveRoom_InputVariable"
          outputVariable="BookRoom_2_reserveRoom_OutputVariable"/>
        […]
      </sequence>
    </catch>
  </faultHandlers>
  <sequence>
    […]
    <invoke name="BookRoom_1" partnerLink="HotelService_1"
      portType="ns1:HotelService" operation="reserveRoom"
      inputVariable="BookRoom_1_reserveRoom_InputVariable"
      outputVariable="BookRoom_1_reserveRoom_OutputVariable"/>
    […]
  </sequence>
</scope>
```

Table 7.2.1 Apache ODE communicationFault

Once the proposed solution is done, the problem is focused in how it can be achieved. They are two main problems:

- How to define this new fault,
- Where it should be placed into the huge ODE source code.
7. Proposed Solution

7.2.1 Fault Definition

Within a client or service implementation, the exceptions are used to communicate errors locally. On the other hand, faults communicate errors services boundaries, for example, from the server to the client.

As was show before, transport channels used to have, in addition, specific mechanisms to communicate transport errors (i.e. the status codes used by HTTP to communicate problems during the transport 4XX, 5XX).

When an exception is thrown, it is very important that it provides enough information to make understand the process what is wrong, how important is this failure, and suggestions of how to fix it.

For our fault definition we can use as reference the standard faults defined in ODE (joinFailure, conflictingReceive, conflictingRequest…) and at the same time this may reduce future problems when we would like to embed it in de ODE source code. This new fault should store and provide enough information about the problem.

```
Class FaultException
    java.lang.Object
    └─java.lang.Throwable
        └─java.lang.Exception
            └─org.apache.ode.bpel.common.FaultException
```

7.2.2 Throwing the Fault

Apache ODE source code contains almost 600 classes and Apache Axis has also a huge number of classes. Management this data is a very big challenge and for this reason this section has to be take only like sketch of reference.

To determine where it needs to implement this new fault is a good philosophy to follow the flow of a call from BPEL process (<invoke>) to a Web Service and discovering the specific moment in that the call to generic AxisFault occurs. Is at this moment when the AxisFault should be replaced by our new “communicationFault”.

Taking as reference the indications specified in [33] and following carefully the source code during the different calls, can be observed that the critical point occurs at the function ”handleResponse” within the class HTTPSender in “org.apache.axis2.transport.http” package by Axis2 (See Figure 7.2.1).
Figure 7.2.1 BPEL <invoke> activity flow
In that function is handled the response received from HTTPClient. After receiving the response the HTTP header is read with the aim of seeing the status, successful or not, of the response.

If the call to Web Server has been satisfactory the code of the HTTP header will be 200 OK and the message will be treated as a good response by Axis2 and ODE respectively.

In case that a fault had occurred in communication, the HTTP header will contain the corresponding error code (4XX, 5xx). At this moment can not known yet if this is a failure in transport, due to, as explained in the previous chapters, it is possible that could be is a SOAP fault which is bound in a HTTP error message: "HTTP 500, Server Error".

The "handleResponse" function examines if the current situation is a case of a SOAP fault or transport fault:

In the first case calls the "processResponse" function within the class AbstractHTTPSender in org.apache.axis2.transport.http package Axis2 package.

If is not the case of a SOAP fault, is because is a transport fault with the corresponding HTTP error code.

Is in this moment in which Axis throws a generic AxisFault, and is here where we must switch that generic fault by our new fault "communicationFault". Table 7.2.2 shows a sketch of how it should looks like:

```java
private void handleResponse(MessageContext msgContext,
                              HttpMethodBase method) throws IOException {
    int statusCode = method.getStatusCode();
    if (statusCode == HttpStatus.SC_OK) {
        processResponse(method, msgContext);
    } else if (statusCode == HttpStatus.SC_ACCEPTED) {
    } else if (statusCode == HttpStatus.SC_INTERNAL_SERVER_ERROR ||
               statusCode == HttpStatus.SC_BAD_REQUEST) {
...
```
-- [ If is a SOAP Fault]
processResponse(method, msgContext);
...
-- [If is a Transport Fault 400 or 500]
throw new ODEFaultException(new QName (qncommunicationFault),
   "Communication Fault:" + String.valueOf(statusCode) +
   method.getStatusText());
} else {
   -- [Transport Fault, except 400 or 500]
   throw new ODEFaultException(new QName (qncommunicationFault),
      "Communication Fault:" + String.valueOf(statusCode) +
      method.getStatusText());
}
}

|Table 7.2.2. Proposed handleResponse sketch |

From this moment ODE proceeded to collect and treat the fault as one of the standard faults that it has defined. Such standard faults are listed in "BPELCompiler" and "OConstant" (ODE packages org.apache.ode.bpel.compiler and org.apache.ode.bpel.o respectively) and is in these places where should be added our "communicationFault".

Once these changes are done the BPEL process should be able to catch our new fault and treat it as its defined standard faults are treated.

In this new situation (Figure 7.2.1), if there were any communication fault along the calling to a Web Service by an <invoke> activity, it would be catch in the <catch> activity by a "communicationFault". The new result is the obtained by the alternative path (<invoke> to the "Hotel Service 2" inside the "faultHandler").
8. Conclusion and Outlook

From the beginning, this thesis was oriented to accomplish a description of the different faults that can be raised across the Web Service stack. At the same time have been described how these faults are reflected, structured and in which error situations they are thrown.

Web Services are, at their essence, distributed applications. Along a communication with an external service, a huge number of different error situations can raise due to the not absolutely reliable communications. L. Peter Deutsch published the “Eight Fallacies of Distributed Computing” [5], which denotes the reasons of why this kind of situations can occurs.

As have been seen during the realization of this thesis, the number of faults that can be raised across the Web Service stack is considerable. In each layer of the stack its faults have a lot of information about what happened in order to provide it to fix the problem. Those faults are passed bottom-up through the Web Services Stack to the upper layers until one is able to manage the fault. One problem observed is that some of the information about the fault is lost in this process, such for example when a HTTP fault is mapped to a generic AxisFault.

Workflow programs include a lot of components and this made that the realization of this thesis was a big challenge, also problems with versions incompatibilities between these programs made the realization a bit more complicated.

Nowadays, there are many BPEL engines; some of the most relevant are Apache ODE, Oracle BPEL Process Manager and ActiveBPEL. To achieve this thesis have been used the first two, the third one has also been used but not in depth and was decided to do not include in this document.

The two engines used share most of their behaviours with the exception that Oracle BPEL Process Manager offers certain predefined runtime faults, which give it some advantages in relation to Apache ODE while dealing with communication faults.

These advantages of Oracle BPEL Process Manager are the ones that would like to extrapolate to Apache ODE engine, defining a new fault which can manage communication faults (mapping them until BPEL process), such as in Oracle BPEL “RemoteFault” and “BindingFault” manage them.

Oracle BPEL Process Manager is not open source and for this reason was not possible to go deep through the code to know how these runtime faults are implemented with the aim of applying this to the Apache ODE engine.
The proposed solution to make possible that the communication faults can be mapped through the Web Service stack until the BPEL process, is the definition of a new fault which collect all the necessary fields for these communication faults and map it to BPEL without lost information about the fault. Also that such fault can be caught by the BPEL engine and treated it within the respective "faultHandler".

The huge source code of Apache ODE and Axis2 managed for the proposed solution of defining a new fault, which could have the same behaviour against communication faults than the two Oracle BPEL runtime faults, is more theoretical than based in the practice.

Future lines of investigation could focus on the implementation and integration of this new defined communication faults into the Apache ODE source code with the aim of being able to deal with this kind of faults like Oracle BPEL Process Manager does.
List of Figures

Figure 2.1.1: OSI Model stack vs. TCP/IP stack ........................................ 11
Figure 2.1.2: Communication between layers by header addition .................. 15
Figure 2.2.1: Web Services stack ............................................................... 16
Figure 2.3.1: OSI/Web Services stack ......................................................... 19
Figure 3.1.1: Travel Agency Example .......................................................... 21
Figure 3.1.2: Travel Agency BPEL representation ......................................... 21
Figure 3.1.3: ICMP Destination Unreachable ................................................. 23
Figure 3.2.1: Example retransmission ......................................................... 24
Figure 3.3.1: Http Server Unavailable .......................................................... 26
Figure 3.3.2: Requesting SOAP node state transitions ................................... 27
Figure 3.4.1: SOAP/HTTP Request/Response .............................................. 30
Figure 4.1.1: Web Services stack: Transport Layer (TCP/IP) ......................... 31
Figure 4.1.2: TCP data transfer with timeout errors ....................................... 40
Figure 4.2.1: Web Services stack: Transport Layer (HTTP) .......................... 41
Figure 4.3.1: Web Services stack: Messaging Layer (SOAP) ......................... 45
Figure 4.4.1: Web Services stack: Description and Discovery (WSDL) .......... 54
Figure 4.5.1: Web Services stack: Business Processes (BPEL) ..................... 57
Figure 4.6.1: WS-Stack Faults ................................................................. 59
Figure 4.6.2: Application (HTTP) error ....................................................... 61
Figure 4.6.3: Messaging Layer error ......................................................... 62
Figure 4.6.4: WSDL fault message ............................................................. 64
Figure 4.6.5: BPEL error message ............................................................ 65
Figure 5.1.1: Travel Agency Example with faultHandler .............................. 68
Figure 5.1.2: Travel Agency with faultHandler: BPEL representation ............. 69
Figure 7.1.1: Communication Faults in the motivating example ..................... 79
Figure 7.1.2: Example Flow without faults .................................................. 81
Figure 7.1.3: Example Flow without define <catch> activity .......................... 82

Figure 7.1.4: Example Flow with “RemoteFault” and “BindingFault”
defined within the <catch> activity.............................................. 86

Figure 7.2.1: BPEL <invoke> activity flow ............................................. 89
List of Tables

Table 3.4.1: Requesting SOAP node transitions .............................................. 27
Table 3.4.2: SOAP/HTTP error ........................................................................ 28
Table 3.4.3: SOAP fault MustUnderstand ......................................................... 29
Table 4.1.1: ICMP type 3 messages ................................................................. 32
Table 4.1.2: ICMP type 11 messages ............................................................... 33
Table 4.1.3: ICMP type 5 messages ................................................................. 34
Table 4.1.4: TCP to Upper-layer primitives ..................................................... 38
Table 4.3.1: SOAP Fault elements ................................................................. 47
Table 4.3.2: SOAP Fault codes ..................................................................... 47
Table 4.3.3 SOAP predefined roles ............................................................... 48
Table 4.3.4 SOAP Binding extends WSDL .................................................... 51
Table 4.3.5: SOAP embedded in HTTP Request ........................................... 52
Table 4.3.6: SOAP embedded in HTTP Response .......................................... 53
Table 4.3.7: SOAP fault VersionMismatch .................................................... 53
Table 4.4.1: WSDL Fault definition ............................................................... 55
Table 4.4.2: WSDL Fault description ............................................................. 55
Table 4.4.3: WSDL binding .......................................................................... 55
Table 4.4.4. SOAP Fault from WSDL binding .............................................. 56
Table 4.6.1: Transport error messages ............................................................. 60
Table 4.6.2: HTTP error messages ................................................................. 62
Table 4.6.3: SOAP Faults cond .................................................................... 63
Table 5.1.1: BPEL fault handler sketch .......................................................... 67
Table 5.1.2: Travel Agency fault handler activity BPEL sketch .................... 69
Table 6.1.1 Oracle BindingFault faults codes .............................................. 74
Table 6.1.2 Oracle RemoteFault faults codes .............................................. 74
Table 7.1.1: “RemoteFault” and “BindindFault” applied to our example........... 80
Table 7.1.2: Oracle VersionMismatch................................................................. 82
Table 7.1.3: Oracle Timeout ........................................................................... 83
Table 7.1.4: Oracle WSDLReadingError .......................................................... 84
Table 7.1.5: Oracle 503 Service Unavailable ...................................................... 85
Table 7.2.1: Apache ODE communicationFault .............................................. 87
Table 7.2.2: Proposed handleResponse sketch ............................................. 90
References


[10] SOAP Version 1.2; http://www.w3.org/TR/soap12-part1/


[13] Web Services Description Language (WSDL) 1.1, W3C Note 15 March 2001; http://www.w3.org/TR/wSDL


[22] RFC-1122. Requirements for Internet Hosts – Communication Layers; R. Braden, Editor; October 1983.


[31] Apache Axis2; http://ws.apache.org/axis2/


[37] SOAP Version 1.2 Usage Scenarios; W3C Working Draft 26; June 2002

All links were last followed on July, 14th 2008.
Declaration

All the work contained within this thesis, except where otherwise acknowledged, was solely the effort of the author. At no stage was any collaboration entered into with any other party.

(Pablo Fuentetaja Abad)