An Approach to Exception Handling for Service-Oriented Systems
Yue Qiang  Wang Hao  Zha Li  Li Wei  Xu Zhi-wei
Research Center for Grid and Service Computing,
Institute of Computing Technology, Chinese Academy of Sciences, Beijing, 100080, P.R. China
{qiangyue, wanghao, char, liwei, zxu}@ict.ac.cn

Abstract

This paper sets out key issues of exception handling that relate to identifying, analyzing and dealing with an exception in service-oriented systems. Then required concepts, basic structures and algorithms for the MI-Eh are discussed. The new concepts include the super-space of exception definitions, the meta-definition mode, and components in an exception message. The main characteristics of the MI-Eh are a uniform definition mode, dynamical extensibility, and abilities of configurable analyses and encapsulations, and a message-level combination. Finally, implementation and evaluation of the mechanism is given.

Key words — message level exception handling (MI-Eh), definition space, meta-definition mode, algorithm

1. Introduction

In service-oriented systems, an exception is an event that occurs during the execution of a message exchange that disrupts the normal flow of messages. WSDL provides a mechanism by the way of <wsdl:Fault> to declare an interface fault component [1][2]. It describes a fault that may occur during invocation of an operation of the interface [2]. For an interface fault component, a mapping to a SOAP fault must be described [3]. In fact, the interface fault component provides a clear mechanism to name and describe the set of faults [2]. This allows operations to easily identify the individual faults they may generate by name [2].

A fault may cause an error. In general, if any application level error occurs in processing a request message UDDI nodes may return specific information within a dispositionReport response to describe an API error [1][4]. Faults that contain disposition reports can be used to determine the cause of an error. Similarly, a dispositionReport element will be return to the caller within a SOAP fault message [3]. Furthermore, error codes can decrease the impact on client interoperability [4].

A Service-Oriented Architecture (SOA) makes it easy to deliver new business applications in the form of service composites or business processes that orchestrate multiple business services (e.g. WS Choreography) [1][5][6]. In order to gain visibility and diagnosis for individual service transaction and multi-step process flows in real-time, some companies (e.g. Sonic, AmberPoint) apply to exception management technologies by catching all types of exceptions midstream and resolving them before they result in lost revenues [5][6].

Very often the exceptional features are not compatible with exception handling features of the programming language they are employed, thus creating a gap that can cause mistakes or unnecessary complications, e.g. the absence of useful exceptional information in service-to-service invocations, or nearly duplicate codes for exception handling. A problem arises in the case how to describe general system exceptions, e.g. input/output errors, invalid message formats etc., which may be generated as part of the message exchange or service implementations. It is unsuitable to describe such general system exceptions in each interface component, which can reasonably be expected to occur. In fact, the management technologies of tracking, monitoring and inspecting exceptions perform their duties, but they could not provide a mechanism to declare or define above general system exceptions in SOAs.

More and more processing layers make SOAs increasingly complex. It is hard to anticipate and handle various processing layer’s exceptional conditions in systems. The developers must ensure that exceptions do not cause problems for users. For users, correct or appropriate exception and error information is needed but too much information is just as much of a problem as too little. Another problem is how to hold exceptions and hide unnecessary technical details for users.

Most development environments equipped mechanisms to handle exceptions and can log the occurrence of exceptions [7][8]. However, it becomes more difficult to manually track business flowing through systems with multiple services coupled together because pieces of exceptional information may be scattered across multiple log files, across multiple servers. The third problem is how
to immediately deliver a group of useful information of remote exceptions to an invoker in a situation of multiple service invocations.

The rest of this paper consists of the following topics. Section 2 discusses key issues about exception handling in service-oriented systems. Section 3 gives the concepts required for the MI-Eh. Basic structures and algorithms of the MI-Eh are given in section 4. Section 5 represents implementation and evaluation of the MI-Eh. Conclusions are given in section 6.

2. Key issues and objective

In the early 1970’s John Goodenough defined the concepts for exception handling models, techniques, and applications. He defined exceptional conditions as follows: “Of the condition detected while attempting to perform some operation, exception conditions are those brought to the attention of the operation’s invoker [9].” In developing service-oriented systems, the existing mechanisms have following problems:

(a) Lack of a uniform mode to declare exceptions. Too many exception classes that inherit directly or indirectly from the base exception class and exception declarations by different modes are difficult to identify because one module or a processing unit often do not recognize properly the exception thrown by others.

(b) How to process a set of exceptional information in order to satisfy different requirements. It includes three sub-questions:
   (i) How to hide a number of unnecessary technical details for a local invoker. Because of increasing processing layers and invoked functions, the technical details that can be fetched are often too large to analyze the cause of an exception immediately and track it properly.
   (ii) Absence of useful information related to a remote occurrence exception. In the situation of multiple service invocations, the useful information should be delivered to the invoker.
   (iii) In addition, when a midstream exception occurs during delivery of a previous exception message, a new exception message maybe cause to lose the previous.

The MI-Eh applies to declaring, generating and dealing with exceptions in service-oriented systems. The objective of MI-Eh is as follows:

(a) The definition of an exception in MI-Eh should be universal uniform. The mechanism could prevent exception declarations from the infection of too many hierarchical exception classes and nearly duplicate codes for exception handling.

(b) The mechanism should provide capabilities to dynamically extend exception declarations. It can describe exceptions of all processing layers in service-oriented systems and hold the intrinsic exception handling mechanisms of built-in components possibly.

(c) The mechanism should provide capabilities to customizing analyze and combine a group of exceptional information. It could automatically hide unnecessary technical details for different levels, handle midstream exceptions and help the invoker easily and quickly identify and track an occurrence exception through delivering one or a group of exceptional information.

A complication is that several exceptions can be raised concurrently in cooperative operations. Exception propagation in operations may not simply track, but require an extra dimension of propagation (e.g., buffers, a stable storage, a local or remote log) [1][8]. This paper does not discuss about such collaborative exception handling [1][7][8] in service-oriented systems.

3. Concepts required for the MI-Eh

To deal with an increasing number of exceptions, the MI-Eh provides a super-space to describe exceptions and defines an exception in a meta-mode. This meta-definition could be employed to describe exceptions that will occur in almost any layer of service-oriented systems. To help the invoker to identify and deal with an occurrence exception, the MI-Eh should provides useful components in an exception message, including the exception identifier, the elements of a matched trace, and a group of other optional information.

3.1. Definition spaces and segments

The super-space for exception definition may consist of a number of definition spaces (including the main space and extended spaces). Each space could be divided into one or many linear segments for grouping the exception definitions.

A definition space in the super-space is a tuple $DS = (PDS, PSGname, DSName)$, where:
- $PDS$: a parent definition space,
- $PSGname$: a segment name in the parent space,
- $DSName$: the name of a definition space.

The $PDS$ and $PSGname$ elements of the main space are always empty. For example, the main space could be defined as:

$\langle null, null, "http://org.ict.vega.gos.exception/" \rangle$.

And an extended definition space could be:

$\langle \{null, null, "http://org.ict.vega.gos.exception/" \}, "User-Defined", "Hotfile" \rangle$.

It means there is a space with the name of “Hotfile”, which extends the main space at the “User-Defined” segment.
A segment in a definition space is defined as $SG = (DS, SGName, MItemList, Begin, End)$, where:
- $DS$: a definition space,
- $SGName$: the segment name,
- $MItemList$: a list of matching items that could be configured by a developer at the phase of initializing the segment,
- $Begin$: the first exception number of this segment,
- $End$: the last exception number of this segment.

For example, the common segment in the main space could be defined as follows:

$\{(null, null, "http://org.ict.vega.gos.exception/"), "Common", \{("java, org.ict.vega"), 0x00, 0xFF\}\}$

In this segment, there are two matching items, "java" and "org.ict.vega". The usage of these items will be discussed in the next section.

The super-space will theoretically contain all definitions of exceptional conditions in service-oriented systems. The super-space should satisfy following criteria:

(a) It is divided into one or more definition spaces by different processing layers or subsystems. Generally, the only one main space is applied to exception definitions in the core layer, extended spaces to others.
(b) The main space has a reserved segment for definitions of common and general system exceptions.
(c) According to various modules or services, each space is divided into linear segments. A segment has a configuration for analyzing elements of exceptional information.
(d) Each segment may choose one exception definition (always the last one) for extending one or many definition spaces.
(e) Each implementation of a module or a service could dynamically map its exception definitions in a segment to exception declarations at initialization, and have a unique path for searching an exception declaration at beginning of the main space.

Therefore, any exception of processing layers in service-oriented systems can be dynamically declared and searched in the super space and its segments.

### 3.2. The meta-definition mode

The meta-definition mode of exceptions is independent of concrete programming languages and mechanisms of exception handling.

An exception meta-definition is a tuple $ExMD = (DS, Num, ExDesc, ExType, UHandler)$, where:
- $DS$: a definition space,
- $Num$: the integer assigned to an exception,
- $ExDesc$: the description of an exception,
- $ExType$: the type of an exception. This paper only discusses synchronous, internal and terminate exceptions and errors.

- $UHandler$: reserved for the future use.

For Example, in the common segment, java language exceptions and errors are located in $0x10-0x4F$. The java.sql.SQLException class and its subclasses are translated into $0x3E$ and its $ExDesc$ getting from the $getMessage()$ method when one occurs. That may be represented as:

$\{(null, null, "http://org.ict.vega.gos.exception/"), 0x3B, getMessage(), 03, null\}$

In fact, the MI-Eh imaginary maps such an instance of the java exception class into its reserved common segment, when one exception actually occurs at run-time. The subclasses of a java exception class can be distinguished from different descriptions. Within the meta-definition mode of exceptions, the universal unique declaration of an exception holds true. This mode can describe exceptions in a definition space either imaginary or really. Moreover, the above problem of too many exception hierarchies does not happen in this way.

### 3.3. Components in an exception message

Service-Oriented systems do not exhibit abstractions like classes, objects, methods, remote procedures, but are based around the concept of message processing [1][3][7][8].

A message of a single exception is $ExMsg = (ExID, ExInfo)$, where:
- $ExID$: a formal identifier of $ExMD$ defined in the section 3.2,
- $ExInfo$: components of exceptional information contained in an occurrence exception and the current message context.

A set of components of exceptional information is defined as $ExInfo = (MSTrace, MID, PUDesc, MContent, EMInfo)$, where:
- $MSTrace$: a group of elements in the stack trace that matched the configured conditions,
- $MID$: an optional description of a related message identifier and the endpoint address where the exception occurs,
- $PUDesc$: an optional description of a processing unit related to the occurrence exception,
- $MContent$: an optional content in the current message context,
- $EMInfo$: components of an embedded exception message. An exception that occurs in a midstream for delivering a message of a previous occurrence exception may cause to lose the message. The MI-Eh provides an $EMInfo$ component to encapsulate the previous exception message.

For multiple invocations, a group of messages must be orderly combined. A combination of messages is defined as $MultExMsg = (\cup ExID, \cup ExInfo)$. The combining
structure with encapsulating tags is given in the section 4.4.

4. Structures and algorithms in the Mi-Eh

This section discusses main structures and algorithms in the Mi-Eh, including maps of the super-space, static and dynamical initializations, configurable analyses and encapsulations, and tags of an exception message.

4.1. Relative maps

In order to build the super-space, there are some maps of definition spaces and declarations of meta-defining exceptions.

An index hash map points to the maps of local definition spaces. A key of the index hash map is an index number of a definition space, and the value is a tuple of that definition space with a map for its segments defined as section 3.1. A local index hash map points to the maps of all definition spaces that are extended from the last exception declaration of a segment. A key of the segment map is a segment name, and its value is the structure of a segment definition together with a reference number, including its MItemList, Begin and End value. The figure 1 illustrates relationships between maps in the super-space.

There are multi-level maps that contain objects of common and meta-defining exceptions in segments. In the first-level map that points to the main space, a key is the integer (Num) of an exception meta-definition defined as section 3.2 and its value is the exception declaration. The main space (MS) is divided into S01 to S0k segments. The directly extended space (ESa) roots the segment S0i in the main space and it is divided into S11 to S1m segments. The indirectly extended space (ESb) roots the segment S1j in the ESa and it is divided into S21 to S2n segments. Given the tuple of a definition space and the integer assigned (Num) to an exception, there is a unique path to find the exception declaration from the main space map to the definition space map.

4.2. Static and dynamical initializations

In the Mi-Eh, common exception declarations are often initialized statically, which are located at a reserved segment in maps of the main space and spaces extended from this segment. All processing units and services share the declarations, e.g. errors of HTTP and SOAP [3][11].

The other exception declarations related to different processing layers of systems or a service are mapped dynamically to a segment or marked by its reference number when they are already mapped. The dynamically initialized segment is always released as a service exits normally or abnormally.

At the initialization of a new definition space, the initializing function increases the current size of the index hash map as a new key, and then set its value to a tuple of the definition space. At a new segment initialization, the function matches the definition space and sets its value. The reference number increases when another module or processing unit initializes the same segment again and decreases when it releases it. When a reference number is zero, the Mi-Eh destroys all exception declarations in the segment.

4.3. Configurable analyses and encapsulations

When an exception implicitly or explicitly occurs or has been thrown out a block or a function, the exception information should be analyzed to hide unnecessary details for an invoker. For a remote invoker, the useful components related to an exception should be encapsulated together. In Mi-Eh, the procedure of encapsulations is as follows:

Step 1: Catch an occurrence exception,
Step 2: Determine whether or not it is analyzed, if it is, go to Step 6,
Step 3: Initialize a new structure to encapsulate the occurrence exception,
Step 4: Search or identify the exception declaration in the map of its definition space, and set the exception identifier (ExID),
Step 5: Call the matching procedure to analyze the stack trace and set MSTrace with the matching one,
Step 6: If there already exists an out fault message in the message context, a midstream exception occurs and then set it to the embedded exception message (EMinfo),
Step 7: If the optional components of exception information can be set, and then set MID, PUDesc, and MContent in terms of the current message context, or else go to Step 9,
Step 8: Generate an encapsulating exception,
Step 9: End encapsulations.
The figure 2 shows the flowchart of encapsulating an occurrence exception in the MI-Eh.

![Flowchart of encapsulating an occurrence exception in the MI-Eh](image)

**Figure 2. The flowchart of encapsulating an occurrence exception in the MI-Eh**

According to the list of matching items in a segment, the handler could analyze the stack trace with an occurrence exception when catching an unanalyzed exception, and construct elements of a new stack trace (\(\text{MSTrace}\)). The algorithm of matching procedure is as follows:

1. Get the list of matching items (\(\text{MItemList}\)) from the segment of the exception identifier (\(\text{ExID}\)).
2. Get and buffer the stack trace where an unanalyzed exception occurs.
3. Read the first element in the stack trace and assign it to variable \(X\).
4. Read the first item in the list. If it is not null, and then assign it to variable \(Y\), or else go to Step 10.
5. Compare \(X\) with \(Y\). If they are matching, and then continue, or else go to Step 7.
6. Push \(X\) in the new stack trace and then go to Step 8,
7. Read next item in the list of matching items. If it is not empty, and then assign it to \(Y\) and go to Step 5, or else continue,
8. Read the next element in the buffered stack trace. If it is not empty, and then assign it to \(X\) and go to Step 4, or else continue,
9. Return the new stack trace,
10. End the matching procedure.

**4.4. Tags of an exception message**

The useful components related to a remote exception should be delivered through the SOAP fault to the invoker. Tags with a sequence number (\(Sn\)) mark components of an exception message. The sequence number is calculated at both service and invoker sides. Components with their tags and their combination can be made up as follows:

(a) If the occurrence exception is not encapsulating, set \(Sn\) to 0, and then give its description, or else set \(Sn\) to 1.
(b) A group of \(<\text{Faultcode}\>\text{Sn}\):\(<\text{Faultstring}\>\text{Sn}\): tags, where the content within a Faultcode tag is in a following form:
   \(\{"\} + \text{DSName} + \{"\} + \text{SGName} + \{:" + \text{Num} + ":\} + \text{ExType}\).
   The descriptions of \(\text{DSName}\), \(\text{SGName}\), \(\text{Num}\) and \(\text{ExType}\) are defined in the section 3.1 and 3.2.
(c) A group of \(<\text{Exceptioninfo}\>\text{Sn}\): tags, the content within an Exceptioninfo tag is a single structure of exception information with sub-tags, \(<\text{WST}>\) for \(\text{MSTrace}\), \(<\text{WMID}>\) for \(\text{MID}\), \(<\text{WPUDESC}>\) for \(\text{PUDesc}\), \(<\text{WMCONTENT}>\) for \(\text{MContent}\) and \(<\text{WEMINFO}>\) for \(\text{EMInfo}\), and contents within these sub-tags refer to section 3.3.

Within these tags and the sequence number, an encoding or decoding procedure could make a combination of exception messages. For an exception occurred in a service side, the encoding procedure translates an encapsulating exception into an exception message. In the invoker side, the decoding procedure converts the exception message into a new exception and appends its identifier and local exception information.

In the situation of multiple service invocations, the MI-Eh guarantees a composite exception message step by step in run-time, and gives the end invoker a whole attentive message of failed invocations.

**5. Implementation and evaluation**

The Vega GOS v2.0 consists of a series of standard web services, developing libraries and grid utilities for deferent users and provide a platform for global sharing resources [10][11]. The main core services include Agora service, Grip (for grid process) service, Grid Router service with exception handling and security mechanisms [10][11]. In the Vega GOS v2.0 distribution, there is another type of services, called system services that contain meta-database management of a grid file system, grid job management, and auditing management of grid jobs [11].

**5.1. An implementation of the MI-Eh**

In the Vega GOS v2.0, the main space map of exception declarations consists of eight segments: Common for the reserved one, Grip, Handler for the security, Router for grid router service, Agora, RMS for the resource management, UMS for the user management and User-
Defined. The main space name is “http://org.ict.vega.gos.exception/”. More details show in the table 1.

The system exception declarations locate in maps of extended definition spaces that root 0x7FFFFFFF, for example, “http://org.ict.vega.gos.exception/Hotfile/” for those in the grid file system, “http://org.ict.vega.gos.exception/Batch/” for those of the batch job service. An exception occurs in a physical service that the invoker does not identify could be encapsulated to a physical service exception (0x0B) in the common segment.

In the Vega GOS v2.0, the main functions of the MI-Eh put into groups of initializing, encapsulating, encoding and decoding. Some APIs are as follows:

1) Initializing APIs
   - initGException(DS ds, int begin, int end): initialize exception definitions in a segment from begin to end. If the ds is null, the segment belongs to in the main space map.
   - addSTMitem(List lt, String str): add an item (str) to the list (lt) of matching items.
   - delSTMitem(List lt, String str): delete an item (str) from the list (lt).
   - uninitGException(SG sg): release the segment of exception declarations.

2) Encapsulating APIs
   - GosException(Exception e, MessageContext mc): encapsulate an general exception to a GOSException, and analyze the original stack trace, and then set the matching stack trace to the GOSException. If the message context (mc) is not null, then set the elements of exceptional information.
   - GosException(DS ds, Num): construct a GOSException to be thrown explicitly.
   - getMessage(): an overridden method of the java.lang.Throwable class [12]. The detail message of a GOSException is one or a group of exceptional messages (ExMsg).
   - toString(): an overridden method of the java.lang.Throwable class. A short description of a GOSException is exceptional identifiers (ExID)
   - printStackTrace(): an overridden method of the java.lang.Throwable class. It prints the detail message of a GOSException to the standard error stream.

3) Encoding and decoding APIs
   - GosE2SOAPFault (GosException ge): encode a GOSException (ge) to a SOAP fault message.
   - SOAPFault2GosException(SoapFault sf): decode a SoapFault (sf) to a GOSException, and append the local exceptional identifier (ExID) and set matching trace elements (MSTrace).
   - The system exception declarations locate in maps of extended definition spaces that root 0x7FFFFFFF. The main space name is “http://org.ict.vega.gos.exception/”. More details show in the table 1.

In the Vega GOS v2.0, the main functions of the MI-Eh put into groups of initializing, encapsulating, encoding and decoding. Some APIs are as follows:

1) Initializing APIs
   - initGException(DS ds, int begin, int end): initialize exception definitions in a segment from begin to end. If the ds is null, the segment belongs to in the main space map.
   - addSTMitem(List lt, String str): add an item (str) to the list (lt) of matching items.
   - delSTMitem(List lt, String str): delete an item (str) from the list (lt).
   - uninitGException(SG sg): release the segment of exception declarations.

2) Encapsulating APIs
   - GosException(Exception e, MessageContext mc): encapsulate an general exception to a GOSException, and analyze the original stack trace, and then set the matching stack trace to the GOSException. If the message context (mc) is not null, then set the elements of exceptional information.
   - GosException(DS ds, Num): construct a GOSException to be thrown explicitly.
   - getMessage(): an overridden method of the java.lang.Throwable class [12]. The detail message of a GOSException is one or a group of exceptional messages (ExMsg).
   - toString(): an overridden method of the java.lang.Throwable class. A short description of a GOSException is exceptional identifiers (ExID)
   - printStackTrace(): an overridden method of the java.lang.Throwable class. It prints the detail message of a GOSException to the standard error stream.

3) Encoding and decoding APIs
   - GosE2SOAPFault (GosException ge): encode a GOSException (ge) to a SOAP fault message.
   - SOAPFault2GosException(SoapFault sf): decode a SoapFault (sf) to a GOSException, and append the local exceptional identifier (ExID) and set matching trace elements (MSTrace).

The scenario is that a client invokes a service, in which an embbed client invokes another service. In the Vega GOS v2.0, when a grip client calls a grip service that invokes a ‘search’ operation undeclared in a physical service, the SOAP message-processing module raises an exception with a parameter of “No such operation 'search'”. The following discussions focus on how to hide the unnecessary details of a stack trace and how to combine the exception message in multiple invocations.

In the grip service side, the SOAP message-processing module firstly invokes the security handler to verify a certificate or proxy certificate of a GOSContext [11]. Because this SOAP fault message received from the invoked physical service does not have a GOSContext, an exception can be caught here.

The original stack trace with the exception has more than 50 lines. Each line is a stack trace element, including a method name, a filename and a line number. Some elements in it are listed as follows with an index number in the parentheses:

1) org.ict.vega.gos.security.handler.impl.
   2) org.apache.axis.client.Call.invoke
   3) org.apache.axis.Strategy.InvocationStrategy.visit
   4) org.apache.axis.SimpleChain.doVisiting
   5) org.apache.axis.SimpleChain.doVisiting
   6) org.apache.axis.SecurityHandlerTransport.init
   7) org.apache.axis.SecurityHandlerTransport.init
   8) org.apache.axis.SecurityHandlerTransport.init
   9) org.apache.axis.SecurityHandlerTransport.init
   10) org.apache.axis.client.Call.invoke

<table>
<thead>
<tr>
<th>No</th>
<th>Definition space name</th>
<th>Segment name</th>
<th>Begin to end</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><a href="http://org.ict.vega.gos.exception/">http://org.ict.vega.gos.exception/</a></td>
<td>Common*</td>
<td>0x00000000-FF</td>
<td>All</td>
</tr>
<tr>
<td>2</td>
<td><a href="http://org.ict.vega.gos.exception/">http://org.ict.vega.gos.exception/</a></td>
<td>Grip</td>
<td>0x000000100-FF</td>
<td>Synch, internal and terminate</td>
</tr>
<tr>
<td>3</td>
<td><a href="http://org.ict.vega.gos.exception/">http://org.ict.vega.gos.exception/</a></td>
<td>Handler</td>
<td>0x00000200-FF</td>
<td>Synch, internal and terminate</td>
</tr>
<tr>
<td>4</td>
<td><a href="http://org.ict.vega.gos.exception/">http://org.ict.vega.gos.exception/</a></td>
<td>Router</td>
<td>0x00000300-FF</td>
<td>Synch, internal and terminate</td>
</tr>
<tr>
<td>5</td>
<td><a href="http://org.ict.vega.gos.exception/">http://org.ict.vega.gos.exception/</a></td>
<td>Agora</td>
<td>0x00000400-FF</td>
<td>Synch, internal and terminate</td>
</tr>
<tr>
<td>6</td>
<td><a href="http://org.ict.vega.gos.exception/">http://org.ict.vega.gos.exception/</a></td>
<td>RMS</td>
<td>0x00000500-FF</td>
<td>Synch, internal and terminate</td>
</tr>
<tr>
<td>7</td>
<td><a href="http://org.ict.vega.gos.exception/">http://org.ict.vega.gos.exception/</a></td>
<td>UMS</td>
<td>0x00000600-FF</td>
<td>Synch, internal and terminate</td>
</tr>
<tr>
<td>8</td>
<td><a href="http://org.ict.vega.gos.exception/">http://org.ict.vega.gos.exception/</a></td>
<td>User-Defined</td>
<td>0x7FFFFFFF0-F</td>
<td>Synch, internal and terminate</td>
</tr>
</tbody>
</table>

5.2. A test of multiple invocations

The scenario is that a client invokes a service, in which an embbed client invokes another service. In the Vega GOS v2.0, when a grip client calls a grip service that invokes a ‘search’ operation undeclared in a physical service, the SOAP message-processing module raises an exception with a parameter of “No such operation 'search'". The following discussions focus on how to hide the unnecessary details of a stack trace and how to combine the exception message in multiple invocations.

In the grip service side, the SOAP message-processing module firstly invokes the security handler to verify a certificate or proxy certificate of a GOSContext [11]. Because this SOAP fault message received from the invoked physical service does not have a GOSContext, an exception can be caught here.

The original stack trace with the exception has more than 50 lines. Each line is a stack trace element, including a method name, a filename and a line number. Some elements in it are listed as follows with an index number in the parentheses:

1) org.ict.vega.gos.security.handler.impl.
   2) org.apache.axis.client.Call.invoke
   3) org.apache.axis.Strategy.InvocationStrategy.visit
   4) org.apache.axis.SimpleChain.doVisiting
   5) org.apache.axis.SimpleChain.doVisiting
   6) org.apache.axis.SecurityHandlerTransport.init
   7) org.apache.axis.SecurityHandlerTransport.init
   8) org.apache.axis.SecurityHandlerTransport.init
   9) org.apache.axis.SecurityHandlerTransport.init
   10) org.apache.axis.client.Call.invoke

...
Obviously, at the grip service side it is difficult to quickly analyze the cause of an occurrence exception and track it properly, because elements of invoking many functions in third party libraries flood the useful elements for a service developer. In this scene, 

GosException or SOAPFault2GosException encapsulating handler can automatically analyze the original stack trace. The list of configured items (MItemList) in the common segment has only one element, the “org.ict.vega” string. The new stack trace consists of elements in the original that matched the “org.ict.vega” string (only 3 lines).

Because this physical service employed the specific mechanism to deal with an occurrence exception other than the MI-Eh, it delivers only one useful faultstring, the ”No such operation 'search’’” description. At the grip service side, firstly the SOAPFault2GosException handler decodes the message, and encapsulates the description together with a new physical service exception (0x0B).

By calling the matching procedure, the handler also filters out unnecessary elements of the local stack trace. Then the GosE2SOAPFault handler encodes the new physical service exception (0x0B) to a new SOAP fault message. Finally, the grip service delivers the new message to the grip client.

All the unnecessary details in a local stack trace (e.g. those of an occurrence exception at the grip client side) are similar to the above. Therefore, the exception handling should hide unnecessary details.

At the grip client side, the decoding, encapsulating (including a matching procedure call) procedure of SOAPFault2GosException handler do like at the grip service side. To call the overridden printStackTrace method the valid components of the SOAPFault2GosException are printed as the figure 3.

With the same java language-defining semantics, it is easy to track the exception by clicking each element in the combination of both local and remote matched traces.

Note: The ExType of a physical service exception (0x0B) is synchronous, internal and terminate (03).

Figure 3. A screenshot for valid components of the exception message in multiple invocations

5.3. Evaluation

The MI-Eh is a generic mechanism of exception handling in service-oriented systems. The implementation of this mechanism can be released as a neutral library, so that the new techniques could be applied to any service or systems if they are executed as default exception handlers.

With studies of some existing mechanisms of exception handling related to service-oriented systems and programming languages, the MI-Eh has following characteristics as being needed for service-oriented systems:

(a) The uniform definition mode. The meta-definition of an exception is global uniform. It is independent of concrete programming languages and mechanisms of exception handling.

(b) Dynamical extensibility. Declarations of exceptions in different processing units of systems and services...
are dynamically extensible.
(c) The ability of configurable analyses and encapsulations. Based on getting all local details and configurable matching conditions related to an occurrence exception, the MI-Eh could automatically hide unnecessary technical details for different levels. The encapsulations could provide necessary components within an occurrence exception and the message context.
(d) The ability of a message-level combination. In multiple invocations, the MI-Eh could orderly combine a set of exception messages in order to deliver useful information.

Because handlers in MI-Eh could have same language-defining semantics, software developers can easily understand those techniques and methodologies of using them.

With the meta-definition mode, it can map almost all exceptions into a super space either imaginary or really, e.g. those of generic systems, services, interfaces, APIs, practical languages, software components. A comparison of some different mechanisms is given in the table 2.

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Scope</th>
<th>Uniformity</th>
<th>Extensibility</th>
<th>Configurable analyses</th>
<th>Combination</th>
</tr>
</thead>
<tbody>
<tr>
<td>MI-Eh</td>
<td>All layers</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>RemoteException</td>
<td>RMI</td>
<td>RMI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WSDL Fault</td>
<td>Interfaces</td>
<td>Interfaces</td>
<td></td>
<td></td>
<td>RMI</td>
</tr>
<tr>
<td>UDDI errors</td>
<td>UDDI APIs</td>
<td>UDDI APIs</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The RemoteException class in java language [13] requires an interface of a remote method that must extend java.rmi.Remote [13], so this specific superclass of Java RMI [13] cannot satisfy needs of exception handling in service-oriented systems that are very often using the non-RMI mode. For a <wsdl:Fault> in the WSDL [2], it is limited to define interface exceptions.

Because handlers of the MI-Eh has a configurable filter function, it could decrease the size of an exception message, comparing with others intend to roughly satisfy requirements for different levels.

6. Conclusions

In this paper, a new mechanism of exception handling is proposed. With concepts of the super-space of exception definitions, the meta-definition mode and components in an exception message, the mechanism could theoretically satisfy different level requirements. Moreover, structures and algorithms in the MI-Eh are verified in the implementation of the Vega GOS v2.0 [11].

As a mention in the introduction, the future work will focus on cooperating handlers for asynchronous events in coordinating services or a workflow [1][7][8]. Nonetheless, the MI-Eh shows a single uniform approach that could satisfy the needs of different-level exception handling in service-oriented systems.

Acknowledgement

This work is supported in part by the China Ministry of Science and Technology 863 Program (Grant No. 2002AA104310), the National Natural Science Foundation of China (Grant No. 69925205), and the China National 973 Program (Grant No. 2003CB317000 and No. 2005CB321807), and the Omii China (Grant No. 2005AA119010).

References