Server-side Exception Handling by Composite Web Services

Kung-Kiu Lau and Cuong Tran

School of Computer Science, the University of Manchester
Manchester M13 9PL, United Kingdom
kung-kiu,ctran@cs.man.ac.uk

Abstract. Currently exception handling for web service orchestrations is performed on the client side. We have defined composite web services [11] that are not single orchestrations but complete web services that contain all possible orchestrations of their sub-services. Our composite web services can therefore define and perform exception handling just once for all such orchestrations, on the server side. In this paper we explain and discuss our approach to server-side exception handling by composite services.

1 Introduction

Currently in web services, client applications are orchestrations of web services provided by various web servers, and exception handling for these applications is defined and performed on the client side (Fig. 1). To be more precise, exception handling is

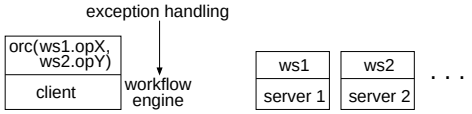

Fig. 1. Client-side exception handling for an orchestration.

performed by the workflow engine on the client side during its execution of an orchestration of web services, e.g. *orc*(*ws1.opX,ws2.opY*) in Fig. 1.

In [11] we defined composite web services that are not orchestrations. An orchestration defines just one workflow for invoking a fixed set of operations, e.g. *opX* in *ws1* and *opY* in *ws2* in *orc*(*ws1.opX,ws2.opY*) in Fig. 1. In contrast, our composite service is a web service offering operations that can invoke all the operations in all its sub-services. In other words, our composite service contains all possible orchestrations of its sub-services.

It follows that our composite service should be able to define exception handling just once for all possible orchestrations of its sub-services. This is clearly an advantage, compared to defining exception handling for one possible orchestration at a time (e.g. *orc*(*ws1.opX,ws2.opY*) in Fig. 1). Moreover, since our composite service is implemented on a server, its exception handling is now performed on the server side (Fig. 2). The benefit of server-side exception handling is that client applications using any or-

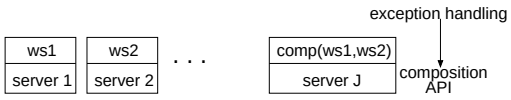

Fig. 2. Server-side exception handling by a composite service.

chestrations contained in the composite service (e.g. *comp*(*ws1,ws2*) in Fig. 2) need not define and perform the exception handling that the composite is already providing. Moreover, it is possible to include recovery actions in a composite service's exception handling, and by so doing, we can make the composite service more stable and reliable from the point of view of all its clients.

In this paper, we show how we define and perform exception handling in composite services on the server side. It is worth noting that our work does not mean to handling all sorts of exception as we shall say clearly in the following sections but to propose a distinct and significant approach to handle exceptions.

There are many exceptions that can be found in web service composition [14]. In our work, we focus on the infrastructure exception *Unavailability*, the process-defined exception *Timeout*, the application exception *Fault*, and suitable recovery actions for these exceptions. There are no obvious, sensible recovery actions for the other exceptions. If the workflow management system fails, there is nothing much we can do about the resulting *Failure* exception. Similarly it is not clear what recovery action is appropriate when the *Delay* or *QoS* exception occurs.

2 Composite Web Services

In our previous work [11], we have defined composite web services. In this section, we give only a brief account of these services. For more details of our composite web services, we would like to advise readers to refer to [11].

We define composite services as distinct from orchestrations. A composite web service *comp*(*ws1,ws2,...*) is a composition of web services (not just their operations, as in orchestrations), where *comp* is a function with the type $comp : ws \times ws \times \cdots \times ws \rightarrow ws$, where *ws* is the type of web services. The composite *comp*(*ws1,ws2,* ...) is thus a whole web service.

This kind of composition is different from an orchestration *orc*(*ws1.opX,ws2.opY,* ...), which defines a workflow for invoking operations in web services. The function *orc* has the type $orc : op \times op \cdots \times op \rightarrow wf$, where *op* is the type of operations in web services, and *wf* is the type of workflows for invoking a set of such operations.

Our composite service thus composes whole services into another whole service. It does so by using composition operators defined in our component model [12, 10].

In our model, components have the distinguishing features of *encapsulation* and *compositionality*. Components are constructed from two kinds of basic entities: (i) *computation units*, and (ii) *connectors* (Fig. 3). A computation unit *CU* encapsulates computation. It provides a set of methods (or operations). *Encapsulation* means that *CU*'s methods do not call methods in other computation units; rather, when invoked, all their computation occurs in *CU*. Thus *CU* could be thought of as a web service.

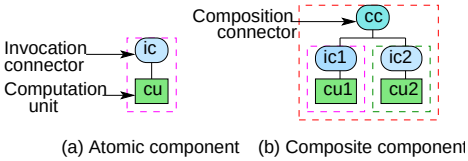

Fig. 3. Our component model.

There are two kinds of connectors: (i) *invocation*, and (ii) *composition* (Fig. 3). An invocation connector is connected to a computation unit *CU* so as to provide access to the methods of *CU*.

A composition connector encapsulates *control*. It is used to define and coordinate the control for a set of components. Composition connectors can be defined for the usual control structures for sequencing and branching. A *sequencer* connector that composes components $C_1, \ldots, C_n$ can call methods in $C_1, \ldots, C_n$ in that order. A *pipe* connector is similar to a sequencer, but additionally passes the results of calls to methods in C_i to those in C_{i+1}. A *selector* connector that composes components $C_1, \ldots, C_n$ can select one component out of $C_1, \ldots, C_n$ and call methods in that component only. The control structure for looping is defined as iterators on individual composition connectors (and invocation connectors, see below).

Clearly composition connectors can define (and encapsulate) *workflow* for a set of connected components. They can define workflow control-flow for sequencing, branching and looping, as described in e.g. [16].

Components are defined in terms of computation units and connectors. There are two kinds of components: (i) *atomic*, and (ii) *composite* (Fig. 3). An atomic component consists of a computation unit with an invocation connector that provides an interface to the component. An atomic component encapsulates *computation*. A composite component consists of a set of components (atomic or composite) composed by a composition connector. The composition connector provides an interface to the composite. A composite component encapsulates *computation* and *control*.[1]

An atomic component can thus be a web service, its invocation connector being the WSDL interface. A composite component can be a (composite) web service that contains sub-services as well as workflow between the sub-services. Its top-level composition connector is its interface. However, this interface cannot be described in standard WSDL since the web service now contains workflow (in the composition connector).

Our components are also *compositional*, i.e. the composition of two components C_1 and C_2 yields another component C_3. In particular, $C3$ also has the defining characteristics of encapsulation and compositionality. Encapsulation and compositionality lead to *self-similarity* of composite components, i.e. a composite has the same structure as any of its sub-components, as can be clearly seen in Fig. 3(b). Self-similarity provides the basis for a hierarchical way of composing systems from components.

We used our model as a component model for web services. We can use standard web services as atomic components, composite web services as composite components, and use the composition connectors as composition operators for web services. This is

[1] We do not consider data encapsulation [15] here, for simplicity.

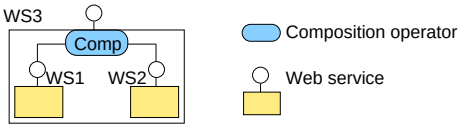

Fig. 4. Composite web services.

illustrated in Fig. 4, where two standard web services *WS1* and *WS2* are composed by a composition operator *Comp* into a composite service *WS3*.

WS3 is a web service, just like *WS1* and *WS2*. However, whereas *WS1* and *WS2* have interfaces described in standard WSDL, *WS3* has an interface that cannot be described in standard WSDL, because *WS3* contains workflow embodied in the composition operator *Comp*. For instance, a *pipe* connector would introduce a workflow structure that connects a number of services, and sequentially invokes each service and uses the result as input to the next invocation; a *selector* connector would provide a branching structure for choosing a service from a set of services, according to a branching or choice condition.

Therefore, in order to define *WS3* as a web service, we need to extend standard WSDL so as to incorporate workflow description. Then we need to devise a method to generate its interface in the extended WSDL from the standard WSDL interfaces of *WS1* and *WS2*. Accordingly, in [11] we defined an extended form of WSDL that contains new <*portType*> elements called <*workflow*>, <*pipe*> and <*choice*> for defining the workflow in a composite service, in a pipe connector and in a selector connector respectively. We also implemented a method for generating the interface of a composite service in extended WSDL from the WSDL interfaces of the composed standard web services.

Example 1. Consider a bank system with just one ATM that serves two bank consortia $B1$ and $B2$, each with two bank branches, $BB1$ and $BB2$, $BB3$ and $BB4$ respectively. The ATM reads the customer's card, performs a security check, identifies the customer's bank consortium and passes customer requests together with customer details to the customer's bank consortium. The customer's bank consortium checks customer details and identifies the customer's bank branch, and then passes on the customer requests and customer details to the customer's bank branch. The bank branch checks customer details and provides the usual services of withdrawal, deposit, balance check, etc.

The bank system can be built as a composite web service composed from standard web services for *ATM*, *B1, B2, BB1, BB2, BB3* and *BB4* (Fig. 5). $P1$, $P2$ and $P3$ are

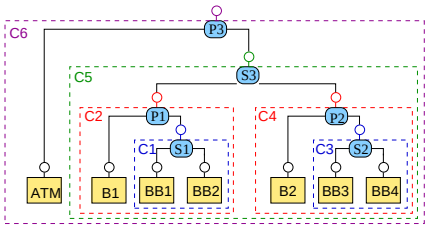

Fig. 5. The bank composite web service.

pipe composition connectors; and $S1$, $S2$ and $S3$ are selector composition connectors. The top-level connector $P3$ is the interface to the system, and is where control flow starts.

The composition is hierarchical (composite services are denoted by dotted boxes): *BB1* and *BB2* are composed into the composite service *C1* by using the selection connector *S1*; the composite *C1* is in turn composed with *B1* using the pipe connector *P1*, creating the composite *C2*; similarly *BB3* and *BB4* are composed into *C3* by using the selection connector *S2*; the composite C3 is then composed with *B2* using the pipe connector *P2*, creating the composite *C4*; the composite *C2* is in turn composed with *C4* by using the selector connector *S3* to create the composite *C5*; the composite *C5* is composed with *ATM* by using another pipe connector *P3*, creating the composite *C6*.

The composite service C6 provides all the operations offered by its sub-services. The workflow of the composite service C6 is outlined as follows:

```
<workflow>
 <pipe> <set name="ATM">
   <operation name="gbc">...</operation> </set>
  <choice>
   <case condition="1">
    <pipe>
     <set name="B1"> <operation name="gbr">...</operation> </set>
     <choice>
      <case condition="1"><set name="BB1">
       <operation name="wd">...</operation>
       <operation name="dp">...</operation> ...</set> </case>
      <case condition="2"><set name="BB2">
        <operation name="wd">...</operation>
        <operation name="dp">...</operation> ...</set></case>
     </choice></pipe></case>
   <case condition="2">
    <pipe>
     <set name="B2"> <operation name="gbr">...</operation> </set>
     <choice>
      <case condition="1"><set name="BB3">
  ...
```

This workflow first invokes the 'get bank consortium' (gbc) operation of ATM, and pipes the result to the branching structure; if the result is 1, then the 'get bank branch' (gbr) operation of B1 is invoked, or if the result is 2 then the 'get bank branch' operation of B2 is invoked. The result of B1's operation is compared with the branching condition; if the value is 1, then *any* one of BB1's operations ('withdrawal' (wd) or 'deposit' (dp)) can be invoked, or if the value is 2, then *any* one of BB2's operations (wd or dp) can be invoked. Similarly the result of B2's operations will lead to the invocation of operations of BB3 and BB4 (not shown here). After that, the workflow ends and the result of the last invocation is returned.

In [11], we gave a detailed implementation of composite web services, as outlined in this section. This implementation provides the basis for our work in this paper.

3 Exception Handling by Composite Services

Our definition of composite services makes it possible to add exception handling to composite services in a hierarchical manner.

In this section, we describe how we can make a composite web service handle common exceptions, as well as provide sensible recovery actions, while invoking its sub-services, which in turn invoke their sub-services, and so on. At each level of sub-services, we shall deal with the following exceptions: *Unavailability*, *Timeout*, *Fault* (see [14]); and we shall provide the following recovery actions: *Retry* and (implicit) *Replace*.

Retry will redo the invocation of the sub-service up to *n* times. If there is a successful response within n times, it is passed back to the caller service. Otherwise, an exception is raised and returned to the caller. The *Retry* action has a default minimum and maximum value for *n*. Without a valid *n* value given by the caller, this default value is used.

Replace and retry will invoke an alternative service to replace the service that has failed. If there is a successful response from the replacement service, it is passed to the caller service. Otherwise, continue to retry on another alternatives until *Retry* reaches the boundary value *n*. Note that, alternative services are chosen by the composite service (implicitly) without caller intervention.

To implement exception handling with these recovery actions, we again use our component model, and integrate the exception handling mechanism into our component model semantics. More precisely, composition connectors (i.e. pipe, selector and sequencer) will intercept all exceptions raised during the invocation of sub-service operations and simply throw them back as normal returns. The whole complex exception handling mechanism will be defined in two new entities added to our component model: *exception guard* and *exception facade*.

3.1 Exception Guard

An *exception guard* is a unary connector that is connected to the interface of a (sub)service (Fig. 6). It receives invocation requests to provided operations in the service and

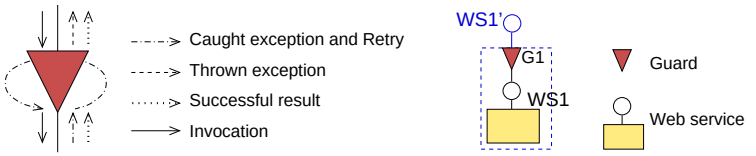

Fig. 6. Exception guard.

intercepts all the results. It captures any exceptions raised by invocations to the service. For exceptions that are *Unavailability*, *Timeout* and *Fault*, it performs the *Retry* recovery action. For other exceptions, it will simply throw them.

An exception guard thus acts as a filter that provides exception handling together with recovery actions. Applying a guard to a service results in a new service, e.g. *WS1* with guard $G1$ becomes *WS1'* in Fig. 6(b). The new service encapsulates the original service and exposes its interface through an extended WSDL document, as we shall see later, to express exception handling semantics. In order to allow the caller a choice of exception handling options, for every provided operation of the original service, we

generate two provided operations for the new service: (i) one operation that invokes the original operation and handles *Unavailability* and *Timeout* exceptions with the *Retry* recovery action; (ii) one operation that invokes the original operation and simply throws any exception encountered. The default and valid range of timeout and retry values used for handling exceptions are preset properties of the exception guard, and are defined in extended WSDL for the new service.

Thus all the provided operations of the original service are made available through the interface of the new service, together with additional exception handling semantics defined by the guard.

For example, in Fig. 6(b), for simplicity, let us assume that *WS1* provides just one operation *op1* that takes one input and returns one output, both as strings. Thus its signature is:

```
op1 (String param) : String
```

The new service *WS1'* provides two operations with exception handling. Their signatures are as follows:

```
op1_RT (String param, String timeout, String num_of_retries) : String
op1_O (String param) : String
```

op1_RT has three inputs: the first one is the input for computation; the second is the value of timeout, and the third one is the number of retries in case of invocation failure. *op1_O* has only one input, which is the input for computation, and whatever exceptions that result will not be handled but simply thrown back.

3.2 Exception Facade

An *exception facade* is an n-ary connector that connects a number of services and produces a new composite service (Fig. 7). The idea behind an exception facade is to unify

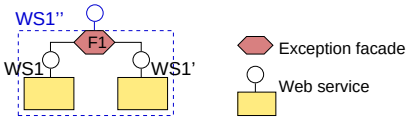

Fig. 7. Exception facade.

two or more services that provide the same operations in order to increase the reliability of operation invocations. If one service cannot respond to a call to one of its operations, then an alternative service is used to replace the failed service. The services are prioritised according to their rankings as backup services, and there is a preset bound on the number of *Retry* actions for the composite service in case of exceptions.

The exception handling behaviour is defined as follows. When there is an invocation to a provided operation, an exception facade delegates the call to the sub-service with the highest priority and intercepts any results. In case of success, the facade passes the result back to the client. In case of exception, the facade recovers by *Retry* actions, provided the bound on retries has not been exceeded.

In detail, if the exception is *Unavailability* or *Timeout*, the facade retries to delegate the invocation to the sub-service with the next highest priority. When there is no more

sub-service with a lower priority, and the number of retries has not reached the bound, the facade starts from the sub-service with the highest priority again. When the number of retries approaches the limit and exception persists, the exception is thrown to the caller. Other exceptions are intercepted but simply thrown back.

As in the case for an exception guard, the default and valid range of timeout and retry values used for handling exceptions are preset properties of an exception facade.

An exception facade thus unifies sub-services and provides an interface for the new service. In order to allow the caller a choice of exception handling options, for each duplicate set of (semantically equivalent) operations provided by the sub-services, two operations are generated for the new service. The interface of the new service will be defined in extended WSDL to expose the generated operations and properties.

For example, in Fig. 7, an exception facade *F1* is applied to two sub-services, *WS1* and *WS1'*, to produce new service *WS1"*. In this example, *WS1'* can be imagined as a backup service for the main service *WS1*, and so their interfaces are identical. *WS1'* is the new service that is reliable and convenient to use, as it provides exception handling for every provided operation.

Again for simplicity, let us assume that the two sub-services *WS1* and *WS1'* have just one operation. The operation accepts one input as a string and return an output also as a string. Thus its signature is:

```
op1 (String param) : String
```

The new service *WS1"* therefore has two operations that have the following signatures:

```
op1_RT (String param, String timeout, String num_of_retries) : String
op1_O (String param) : String
```

op1_RT has three inputs: the first one is the input for computation; the second is the value of timeout and the third is the number of retries in case of invocation failures. *op1_O* has only one input, which is input for computation and whatever exceptions result will not be handled and simply thrown back.

4 Defining Composite Web Services with Exception Handling

A standard web service has its interface described in a WSDL document. Our composite web service has an interface in an extended form of WSDL [11]. Now with exception handling, we are introducing yet more new semantics to composite web services. Therefore, the interface of a composite web service should be further extended with a new element to capture such semantics.

We add one new element, namely <*exception*>, into the conventional <*portType*> element. The <*exception*> element consists of two child elements, <*timeout*> and <*retry*>. Each of these elements has attributes such as *defval*, *minval* and *maxval* to represent default, minimum and maximum value of timeout and retries. The syntax of the extended WSDL is as shown in Fig. 8. For example, the service *WS1'* in Fig. 6 which is built from applying a guard to the service *WS1* will have the extended WSDL interface shown in Fig. 9. The original service *WS1* provides one operation, namely *op1*. The new service *WS1'* has two operations, *op1_O* and *op1_TR*. The *op1_O* operation is identical to the *op1* operation provided by *WS1* while *op1_TR* has an exception

```
<wsdl:portType name="nmtoken">*    <wsdl:documentation .... />?
  <wsdl:operation name="nmtoken">*   <wsdl:documentation .... />?
    <wsdl:input name="nmtoken"? message="qname">?   <wsdl:documentation .... />?   </wsdl:input>
    <wsdl:output name="nmtoken"? message="qname">?   <wsdl:documentation .... />?   </wsdl:output>
    <wsdl:fault name="nmtoken"? message="qname">?   <wsdl:documentation .... />?   </wsdl:fault>
  </wsdl:operation>
  <exception>?
    <timeout defval="nmtoken" minval="nmtoken" maxval="nmtoken">?
    <retry defval="nmtoken" minval="nmtoken" maxval="nmtoken">?
  </exception>
</wsdl:portType>
```

Fig. 8. Extended WSDL for exception handling.

```
<wsdl:portType name="W1'">*
  <wsdl:operation name="op1_O">
    <wsdl:input name="param" message="inMessage">   </wsdl:input>
    <wsdl:output name="return" message="outMessage">    </wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="op1_TR">
    <wsdl:input name="param" message="inMessageTR">   </wsdl:input>
    <wsdl:output name="return" message="outMessageTR">    </wsdl:output>
    <wsdl:fault name="exception" message="faultMessageTR">    </wsdl:fault>
  </wsdl:operation>
  <exception>
    <timeout defval="−1" minval="100" maxval="30000">
    <retry defval="0" minval="0" maxval="5">
  </exception>
</wsdl:portType>
```

Fig. 9. Extended WSDL interface of *WS1'*.

handling mechanism. Invoking *op1_TR* allows us to use exception handling through the parameters of its invocation.

The extended WSDL interface is connected to a composition connector, e.g. a pipe or a selector, in any composition to build further composite services. The interface of such a composite service therefore has a workflow structure (introduced by the pipe and selector connector) as well as exception handling semantics.

Example 2. Consider the bank system in Example 1, as a composite web service (Fig. 5). All the sub-services may be located on different networks; that is, the bank system may be spread over different networks and different platforms. This is where problems arise. In general the bank system can encounter hardware and software problems such as server hardware failure, server software failure, network disconnection or congestion, etc.

For example, if the connection to bank branch service *BB1* is broken, then customers having accounts in this bank branch cannot use the bank services. These problems which are normally unexpected can cause the whole system to malfunction or totally broken down. Thus, this problem must be addressed in order to build a secure and reliable system.

We can define the bank system as a composite service with exception handling, by adding exception guards and facades. The result is depicted in Fig. 10. We assume that all the bank branch services, BB1, BB2, BB3 and BB4 have backup services, BB1', BB2', BB3' and BB4' respectively; services B1, B2 and ATM do not have backup services. To start building the composite bank service, every pair of main service and backup service is first composed with facades F1, F2, F3 and F4 to achieve four reliable

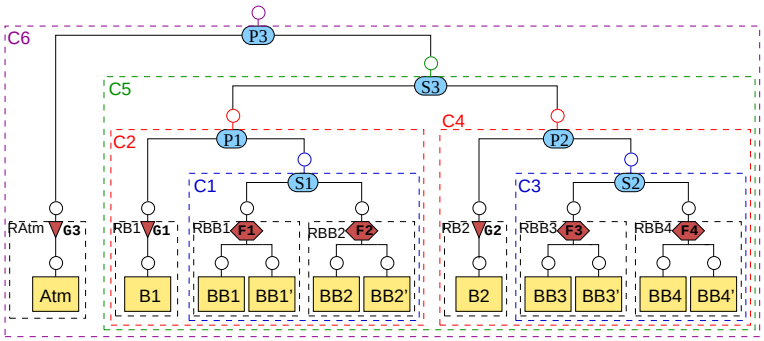

Fig. 10. The bank composite web service with exception handling.

bank branch services RBB1, RBB2, RBB3 and RBB4 respectively. They will then be composed by two selector connectors S1 and S2 to achieve two composite services C1 and C2. B1 and B2 do not have backup service so we apply two exception guards G1 and G2 to them to make two reliable bank services RB1 and RB2, which will be composed with two previously built composite service C1 and C2 respectively, by using two pipe connectors P1 and P2 to achieve two bigger composite services C3 and C4 respectively. We then compose C3 and C4 by using selector S3 to get composite C5. The ATM service also does not have a backup service, so we apply the exception guard G3 to make reliable service RATM, and then compose it with the composite C5 using the pipe connector P3, to get the composite C6. C6 represents the composite bank service with exception handling by providing all services with exception handling.

The interface of the composite C6 is described in extended WSDL, and is outlined as follows:

```
<workflow>
 <pipe> <set name="RATM"> <operation name="getbank_O">...</operation>
                          <operation name="getbank_TR">...</operation>
        <exception> <timeout defval="-1" minval="100" maxval="30000"/>
                    <retry defval="0" minval="1" maxval="5"/> </exception> </set>
  <choice>
   <case condition="1">
    <pipe> <set name="RB1"> <operation name="getbranch_O">...</operation>
                            <operation name="getbranch_TR">...</operation>
           <exception> <timeout defval="-1" minval="500" maxval="30000"/>
                       <retry defval="0" minval="1" maxval="5"/> </exception> </set>
     <choice>
      <case condition="1"><set name="RBB1">
       <operation name="withdraw_O">...</operation>
       <operation name="withdraw_TR">...</operation>
       <operation name="deposit_O">...</operation>
       <operation name="deposit_TR">...</operation> ...
       <exception> <timeout defval="-1" minval="500" maxval="10000"/>
                <retry defval="0" minval="0" maxval="5"/> </exception> </set> </case>
      <case condition="2"><set name="RBB2">
         ...
       <exception>
        <timeout defval="-1" minval="1000" maxval="10000"/>
        <retry defval="0" minval="2" maxval="10"/> </exception>
      </set></case></choice></pipe></case>
   <case condition="2">
    <pipe>
     <set name="RB2">
  ...
```

This workflow first invokes *any* operation of RATM, and pipes the result to the branching structure; if the result is 1, then *any* one of RB1's operations can be invoked, or if the result is 2 then *any* one of RB2's operations can be invoked; the result of RB1's operation is used to compare with the branching condition; if the value is 1, then *any* one of RBB1's operations can be invoked, or if the value is 2, then *any* one of BB2's operations can be invoked. Similarly the result of RB2's operations is used in checking the branching condition; if the condition is 1 then *any* one of RBB3's operations or RBB4's operations will be invoked. After that, the workflow ends and the result of the last invocation is returned. The integrated exception handling mechanism can be used in every invocation of any operation provided by any sub-service in this workflow by selecting the appropriate operation according to its signature, e.g. an operation name ending in *_TR*.

From the point of view of its clients, C6 appears reliable because it handles exceptions with recovery actions. Clients need not repeat the exception handling already performed in C6. They can simply invoke C6 with operations and exception handling mechanisms as parameters, as specified in its interface.

5 Implementation

To implement composite services with exception handling as defined in the previous sections, we make use of our existing implementation of composition connectors (i.e pipe and selector), and extend our existing implementation of composite services by adding the implementation of the exception guard and exception facade connectors. Our existing implementation [11] is in Java and the Axis framework [3].

Applying a guard connector to one service or applying a facade connector to two services results in a (composite) service with an interface in extended WSDL. We need to generate a Java implementation for such an interface. The implementation will dispatch any requests for operations in the interface of the composite to the implementation of the guard or facade connector, and pass any result from the latter back to the client.

Our implementation of the exception guard and the exception facade connector has two (overloaded) operations both called *invoke*. For simplicity, our implementation only deals with parameters of primitive data types, e.g. string, integer, float, etc. We use String as intermediate type because other primitive types can be converted to String and vice versa. In addition, the implementation detects exceptions, e.g. *Unavailability*, *Timeout* and so on, as shown in these signatures:

```
String invoke (String oper, String[] params)
     throws UnavailableException, TimeoutException, Exception

String invoke (String oper, String[] params, int timeout, int retry)
     throws UnavailableException, TimeoutException, Exception
```

Thus the exception guard or the exception facade connector receives an operation, a list of parameters, together with timeout and retry values optionally depending on which operation receives the request. If the first *invoke* operation receives the request, it is invoked with the given list of parameters, and a successful result or any exception will be returned as an output message or a fault message respectively by the composite service. If the second *invoke* operation receives the request, it is invoked with the list of

parameters, and it sets the timeout of this invocation according to the timeout value. A successful result is returned as a normal result. If there is a *Timeout* or *Unavailability* exception, the connector will recover by retrying the invocation to the appropriate service (the same service in the case of a guard; an alternative service in the case of facade). If the exception still persists, it will be returned as a fault message. If the exception is of other types, it will also be returned as a fault message by the composite service.

Although it is expressed in a form of WSDL which is further extended from that in our previous work, the interface of a composite with a guard or facade connector has a top-level <*workflow*> element that is the same as before. As a result, we can use our existing composition connectors in [11], i.e. pipe and selector, on these composites to build bigger composite services. These connector will therefore expose operations provided by sub-services through interfaces to client as already defined in our previous work [11].

Briefly, the Java class of a pipe or selector connector always has one operation *invoke*, that is the operation provided by the composite service to the outside world. Clients use a composite service via its *invoke* operation.

Basically, the signature of *invoke* comprises three main elements, viz. condition, operation names and operation parameters, plus return and exception types. The condition is used in a branching workflow structure for selecting sub-services. Other elements have the same meaning as in the exception guard and exception facade connector.

Each operation is provided by a sub-service which could be either a service with exception handling, or a composite service. If a sub-service is service with exception handling, the connector identifies the number of parameters for every operation from its interface so that the connector can call it with the correct parameters extracted from the parameter list. The result that is either a success or an exception is returned as the output of the composite. If a sub-service is a composite service, the connector just passes the whole operation list at that point to the *invoke* operation of the sub-service.

Finally, in order to demonstrate how our exception handling approach performs in practice, we created a random test case where we simulate *Unavailability* and *Timeout* exceptions. The data for the test case is shown in Table 1, namely services, response times and hosting servers. All the reliable sub-services, RAtm, RB1, RB2, RBB1, RBB2, RBB3, RBB4, and composite services C1, C2, C3, C4, C5, C6 are created and deployed onto one reliable server, because we do not have more servers available, but we can easily extend our test case to many such servers.

Service	Response time	Location
ATM	No delay	Server 1
B1, BB1	Delay 0-1sec	Server 1
BB2	Service moved	Server 1
BB1', BB2'	Delay 0-2sec	Server 2
B2	Delay 0-2sec	Server 4
BB3, BB4	Server down	Server 3
BB3'	Delay 0-1sec	Server 4
BB4'	Delay 0-1.5sec	Server 4
RAtm, RB1, RB2, RBB1, RBB2, RBB3 and RBB4	-	Reliable server
C1, C2, C3, C4, C5 and C6	-	Reliable server

Table 1. Test case data.

Fig. 11 shows the composite bank service handling a balance checking request and the resulting exceptions step by step. When the composite bank service C6 receives a

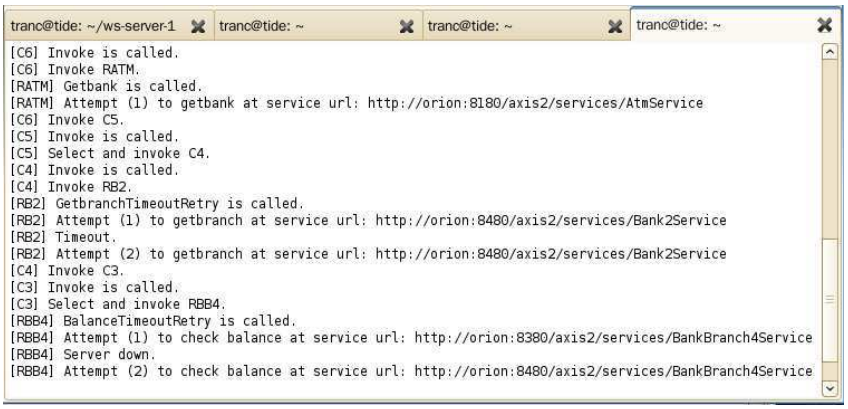

Fig. 11. Test case result.

request, it invokes the get bank operation of sub-service RATM. The result is that bank id is piped to composite service C5. C5 then uses the value to select service C4 which invokes the get branch operation of service RB2. RB2 encounters a timeout exception, and retries to call its sub-service again. The second call of RB2 is successful, and the result is that the branch id is passed to service C3. C3 uses the branch id to select branch service RBB4 to invoke. Again, another exception is encountered and is handled by retrying invocation of another service located at different server.

6 Discussion and Related Work

Our approach offers server-side exception handling by composite web services of some common exceptions. In the literature we have not found an equivalent approach to ours.

The main advantage of our approach is that we define and perform exception handling once, for all the orchestrations contained in a composite service. Existing approaches do not use composite services, but use individual orchestrations, and as a result they have to define exception handling for every orchestration.

For example, let us use the bank example to compare our approach with the *try-catch* approach supported by BPEL for client-side exception handling approach. Suppose there are two clients who want to build their own orchestrations from given atomic (standard) services introduced in our example. One orchestration provides the *withdraw* service and can handle two exceptions, *Unavailability* and *InvalidWithdraw*. The other orchestration provides the *balance checking* service and handles one exception, *Unavailability*. In BPEL, both composition and exception handling are defined at the level of operations. The workflow only composes several specific operations and provides

exception handling for them. Different clients who wish to compose different specific operations of the same services, need to define different workflows together with exception handling mechanisms. Thus, to build the above two applications, each client using client-side exception handling needs to build a workflow with its own exception handling mechanism.

Another advantage of our approach is that a composite service with exception handling is a reliable service, from the point of view of all its clients, whatever orchestration (contained in the composite service) they are using.

So far, all the approaches to exception handling we have found are for the client side. We summarise them here.

[13, 5, 6] present some taxonomies of exceptional situations. [5] also discusses exception handling in workflow management systems. At the lowest level, BPEL [2] provides the primitive try-catch construct to specify the exception to be caught, and define compensation actions as part of workflow specification.

In [8], a processor is built to inject codes (try-catch structures) into a BPEL orchestration, so that the resulting BPEL workflow when encountering exceptions will request for alternative services by invoking an external service.

The approaches in [4] and [13] use the ECA rule for modelling exceptions. The ECA rule in [4] is stored and used by a processor which interacts with the commercial FORO workflow engine to handle exception. In [13], the JECA rule is processed and combined with a CBR (Case-Based Reasoning) mechanism to enhance exception handling. Similarly, ADOME-WFMS [7] also uses ECA to model exception and resolution which is integrated in the ADOME workflow management system.

In [1], an extension to the YAWL workflow engine is created to allow defining exception handling rule sets and associated recovery actions (defined by Exlet) which will be consumed a workflow engine.

In [14, 9], a policy is used to specify what exception can be captured and how exception can be resolved. The policy is either used to generate exception handling construct for the target BPEL workflow [14] or processed by a workflow middleware [9].

7 Conclusion

In this paper we propose an approach to server-side exception handling by composite web services. Currently, our composite services can capture *Unavailability* and *Timeout* exceptions and provide *Retry* as recovery action, or *Throw* to propagate exceptions. Since our composite service contains all possible orchestrations of its sub-services, clients can use any such orchestration without needing to repeat the exception handling already provided by the composite service. As a result, our composite service is reliable, from the point of view of all its clients. Thus our approach offers important benefits to clients for building service oriented applications.

In future work, we intend to investigate sensible resolution for other exceptions such as *Delay* and *QoS Degradation*. This will add even more value to our server-side exception handling approach.

References

1. W. M. P. van der Aalst *et al*. Dynamic and extensible exception handling for workflows: A service-oriented implementation. Technical report, BPM Center, March 2007.
2. T. Andrews *et al*. BPEL4WS - version 1.1. Technical report, IBM, 2003.
3. Apache. Axis - web services framework web site. `http://ws.apache.org/axis2/`.
4. F. Casati, S. Ceri, S. Paraboschi, and G. Pozzi. Specification and implementation of exceptions in workflow management systems. In *TODS*, volume 24, pages 405–451, 1999.
5. F. Casati and G. Cugola. Error handling in process support systems. In A. Romanovsly et al., editor, *Exception Handling*, pages 251–270. Springer-Verlag Berlin Heidelberg, 2001.
6. K.S. May Chan, J. Bishop, J. Steyn, L. Baresi, and S. Guinea. A fault taxonomy for web service composition. In *Proceedings of WESOA07*. Springer LNCS, 2007.
7. D. K. W. Chiu *et al*. Adome-wfms: Towards cooperative handling of workflow exceptions. In A. Romanovsky et al., editor, *Exception Handling*, pages 271–288. Springer, 2001.
8. K. Christos, V. Costas, and G. Panayiotis. Enhancing bpel scenarios with dynamic relevance-based exception handling. In *ICWS*, pages 751–758, 2007.
9. A. Erradi, P. Maheshwari, and V. Tosic. Recovery policies for enhancing web services reliability. In *IEEE International Conference on Web Services (ICWS'06)*, 2006.
10. K.-K. Lau, L. Ling, and Z. Wang. Composing components in design phase using exogenous connectors. In *Proceedings of 32nd ECSEAA*, pages 12–19, 2006.
11. K.-K. Lau and C.M. Tran. Composite web services. In *In C. Pautasso and T. Gschwind, editors,* Proceedings of 2nd Workshop on Emerging Web Services Technology, 2007.
12. K.-K. Lau *et al*. Exogenous connectors for software components. In G. Heineman *et al.*, editor, Proc. 8th Int. Symp. on CBSE, LNCS 3489. Springer, 2005.
13. Z. Luo, A. Sheth, K. Kochut, and J. Miller. Exception handling in workflow systems. In *Applied Intelligence*, pages 125–147. Kluwer Academic, 2000.
14. L. Zeng, H. Lei, and Boualem Benatallah. Policy-driven exception-management for composite web services. In *Proceedings of CEC05*. IEEE, 2005.
15. K.-K. Lau and F. Taweel. Data encapsulation in software components. In *In H.W. Schmidt* et al.*, editor,* Proc. 10th Int. Symp. on Component-based Software Engineering, LNCS 4608, pages 1–16. Springer-Verlag, 2007.
16. W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow patterns. In Distributed and Parallel Databases, pages 5–51, 2003.