Exception Resolution for BPEL Processes: a Middleware-based Framework and Performance Evaluation

Kareliotis Christos
PhD Candidate
Department of Informatics and Telecommunications,
University of Athens,
Greece
+302107275220
ckar@di.uoa.gr

Dr. Costas Vassilakis
Assistant Professor,
Department of Computer Science and Technology,
University of Peloponnese,
Greece
+302710372203
costas@uop.gr

Efstathios Rouvas
PhD Candidate
Department of Informatics and Telecommunications,
University of Athens,
Greece
+302107275220
rouvas@di.uoa.gr

Dr. Panayiotis Georgiadis
Professor
Department of Informatics and Telecommunications,
University of Athens,
Greece
+302107275235
p.georgiadis@di.uoa.gr

ABSTRACT
WS-BPEL has become the predominant technology for specifying and executing composite business processes within the Service Oriented Architecture. During the execution however of such a composite business process, a number of faults stemming from the distributed nature of the SOA architecture (e.g. network or server failures) may occur. To this end, the WS-BPEL scenario designer must exploit the provisions offered by WS-BPEL to catch exceptions owing to system failures and resolve them, typically by invoking some alternate equivalent web service that is expected to be reachable and available. The task of system fault handler specification is though an additional burden for the WS-BPEL scenario designer and the presence of such handlers within the WS-BPEL scenario necessitates additional maintenance activities, as new alternate services become available or some of the specified ones are withdrawn. In this paper, we propose a middleware-based framework for system exception resolution, which undertakes the tasks of failure interception, discovery of alternate services and their invocation. The middleware is deployed and maintained independently of the WS-BPEL scenarios, removing thus the need for specifying and maintaining system faults within the scenarios. We also present performance measures, establishing that the overhead imposed by the addition of the proposed middleware layer is minimal.

Categories and Subject Descriptors
H.3.4 [**Information Storage and Retrieval**]: Systems and Software – *distributed systems* H.3.5 [**Information Storage and Retrieval**]: Online Information Services – *web-based services*; D.2.8 [**Software Engineering**]: Metrics – *performance measures*;

General Terms
Measurement, Performance, Design, Reliability.

Keywords
Web services; Exception handling; Middleware; Performance metrics; Quality of service (QoS), Scalability.

1. INTRODUCTION
Web services are unanimously supported by major software vendors of middleware technology [1]. The main objective of web service technology and related research [2] is to provide the means for enterprises to do business with each other and provide joint services to their customers under specified Quality of Service (QoS) levels. Business Process Management (BPM) addresses how organizations can identify, model, develop, deploy, and manage their business process, including processes that involve IT systems and human interaction.

Business processes are typically complex operations, comprising of numerous individual stages, and in the context of SOA each such stage is realized as a web service. The composition of these steps (control flow, data flow, etc) is frequently specified using the Web Services Business Process Execution Language (WS-BPEL) and executed by a Web Services Orchestration (WSO) platform. Web Services due to their loosely-coupled nature in Service Oriented Architectures (SOAs) provide the flexibility that enterprises need to adapt quickly for satisfying the increased business demands. However, they introduce new challenges when it comes to ensuring superior performance and availability. IT teams lack visibility into web services transactions as they traverse these environments, where services are often shared among several applications and failure (or *exceptions*) can occur anywhere along the transaction path. In this paper we focus on exceptions occurring in business process execution and particularly when a service becomes unavailable; this can be owing to a number of reasons, which may be both transient –e.g. host inoperability, software malfunction, network failure/ partitioning- or permanent –e.g. the service has been withdrawn or changed to some form incompatible to the previous one. In the presence of these events, there is an issue on how to ensure the availability of these services and eventually on how to ensure the health of the business process execution in a real-time production environment. Typically, a replacement component should be identified and substituted for the failed one. The replacement component should have the "same skills" with the failed one i.e. to have same functionality, while some non-functional parameters (e.g. security, performance, response time) can be taken into account [3]. WS-BPEL provides constructs for catching unavailability faults and invoking replacement services through the *Catch* and *CatchAll* activities: the WS-BPEL scenario designer may use these activities to intercept faults and specify which replacement service(s) should be invoked when the "normal flow" service is unavailable. This approach has however the following shortcomings:

1. the WS-BPEL designer must undertake one extra task, i.e. to locate equivalent services and include calls to them into fault handlers within the WS-BPEL scenario.
2. as new services emerge, which might be more suitable as replacements to "normal flow" services than the originally

specified replacements, the related WS-BPEL scenarios need to be maintained. Maintenance activities need to be also taken when replacement services are withdrawn.

Note that due to the static nature of WS-BPEL, which dictates that service bindings should be hard-coded in the scenario, it is not feasible to include calls to all replacement services within a fault handler (typically 2 or 3 alternates will be specified) and not possible at all to dynamically introduce new bindings or remove outdated ones, to align with the changes in service availability. Each such change should trigger a maintenance activity that will lead to modifications in the WS-BPEL scenario code.

In this paper we introduce the Alternative Service Operation Binding (ASOB) framework, which is a middleware-based approach for dynamically resolving exceptions occurring in WS-BPEL scenario executions, elevating the robustness and reliability of business processes and simplifying the maintenance of their specifications. The ASOB framework catches system exceptions occurring in within WS-BPEL scenario execution and resolves them by invoking operational replacement services that are functionally equivalent to the failed ones. ASOB acts as a web service proxy, so its is invoked every time the WS-BPEL engine invokes a service operation. The real invocation is performed by ASOB, and thus ASOB intercepts any faults that occur in this invocation. If a failure is detected, ASOB queries an appropriate registry to identify web services that are equivalent to the failed one, and subsequently invokes them until one of them yields a reply; finally the reply is returned to the WS-BPEL scenario.

The rest of the paper is organized as follows: section 2 presents related work, while section 3 briefly presents the SOA and WS-BPEL provisions used for our purposes. Section 4 presents the overall architecture of ASOB. In section 5 we illustrate implementation specifications and performance results aroused by extended benchmarking when applying ASOB-based BPEL process execution. Finally in section 6 conclusions are drawn and future work is outlined.

2. RELATED WORK

Exception resolution is recognized as an important issue in the context of WS-BPEL scenario design and execution. All WS-BPEL design environments, such as ActiveBPEL [4], Oracle BPEL Process Manager in Oracle Application Server [5], Eclipse [6] OpenESB [7], include provisions through which designers may specify the activities to be taken upon occurrence of some specific (*Catch* construct) or a generic (*CatchAll* construct) fault; these specifications are honored by WS-BPEL orchestrators. Some environments cater for specialized handling of failed requests, e.g. Oracle Process Manager addresses system faults by sending an exception message in a JMS Dead Letter Queue.

The shortcomings of manual fault handler design have become apparent to the industry and the research community alike, and therefore numerous attempts have been made to enhance and/or automate the exception handling process in WS-BPEL scenario execution. [8] presents a methodology and related tools through which various fault tolerance patterns can be mapped to WS-BPEL including provisions for configuring fault tolerant mechanisms on a per-operation basis. This work enhances the original WS-BPEL scenario with fault handlers, employing nested scoping for separating designer-provided fault handlers (usually crafted to tackle *application logic-level* faults -such as insufficient balance while withdrawing from an account- as opposed to *system-level* faults which inhibit the execution of the web service). This technique introduces however the need to either use a specific development environment which will cater for the generation of fault handlers, while it does not also address the issue of introduction of new or withdrawal of existing alternate services (in these cases, the definitions of alternate services should be modified accordingly and the WS-BPEL scenario should be regenerated).

An architecture on how exceptions can be resolved in a generic way is presented in [9], which introduces an additional module, SRRF, which undertakes the handling of exceptions, dynamically discovering services equivalent to the failed one and performing hot-swapping; however the communication of this module with the executing scenario is unclear and details on how exceptions will be directed to the SRRF module or how results of hot-swapping will be returned to the WS-BPEL scenario. [10] elaborates on this approach introducing a pre-processor which enhances the WS-BPEL scenario with fault handlers within nested scopes that redirect faults to the *Alternate WS Locator Module* that returns to the WS-BPEL script the identity of the web service that should be invoked in place of the failed one; however it appears that since WS-BPEL does not allow dynamic service bindings, the pre-processor would need to embed specific calls to alternate services in the produced WS-BPEL script, inhibiting thus dynamic discovery of alternate services and necessitating re-runs of the pre-processor when the list of equivalent services is modified.

In [11] a Web Service Manager for discovering the exception location along the SOA transaction path is presented. This work provides a detailed discussion of CA Wily Web Services Manager and offers concrete examples of how IT teams are using it in the real world to gain control over the performance and availability of web services. Although this work addresses exceptions in composite service execution, it is mainly targeted to pinpointing the exact fault location in environments involving legacy systems, web-based application and other components, while exception resolution is not adequately addressed.

An essential underpinning for the fully automated and dynamic resolution of exceptions during the execution of WS-BPEL scenarios is the ability to locate services which can be substituted for the failed one. A noteworthy approach towards this direction is the one undertaken by METEOR-S project [12], [13] in cooperation with WSMX (Web Services Execution Environment) [14]. WSMX contains the discovery component, which undertakes the role of locating the services that fulfill a specific user request. This task is based on the WSMO conceptual framework for discovery [15]. WSMO includes a Selection component that applies different techniques ranging from simple "always the first" to multi-criteria selection of variants (e.g., web services non-functional properties as reliability, security, etc.) and interactions with the service requestor. Both in the METEOR-S and other approaches, functional and non-functional properties are represented using shared ontologies, typically expressed using DAML+OIL and the latter OWL-S. Such annotations enable the semantically based discovery of relevant web services and can contribute towards the goal of locating services with “same skills” [3] in order to replace a failed service in the process flow. METEOR-S and WSMX also address the issue of exception

resolution exploiting the service equivalence information, they use however pre-determined exception resolution scenarios.

3. SOA PROVISIONS FOR FAULT HANDLING

3.1 Logical Versus System Faults

Business processes specified in BPEL will interact with partner processes through operation invocations on web services. We will refer to these business processes as BPEL processes for the rest of this paper. Loosely-coupled web services in Service Oriented Environments are very sensitive on becoming unavailable for at least a short period of time, since they usually communicate over the internet. Since BPEL processes are -in most cases- long-running transactions, the web services participating in those have to be available and stable anytime. In BPEL processes two kinds of faults can be raised: *logical* and *system*. The first category includes those faults deliberately raised by constituent services to indicate that some form of special handling is required. For example, an *InsufficientCredit* exception thrown by some *CreditCardPayment* service indicates that payment through the credit card is impossible because the credit limit has been exceeded; the BPEL scenario designer may catch this fault type and either end the scenario or attempt to use alternative payment methods, such as direct withdrawal from a savings account or cash payment, if applicable. The second category, namely *system faults*, includes faults not directly raised by constituent services but rather detected by the execution environment. Examples of such faults are the inability to communicate with the hosting server (server down or network partitioning), system-generated responses indicating that the service is not offered at the specific address, parameter number or type mismatches (service has been altered) and timeouts in receiving replies. If a system fault occurs while executing a BPEL scenario, it is possible to remedy the situation by invoking some alternate implementation, since the fact that the particular invocation failed does not imply that other implementations will fail as well (the failure reason is directly bound to the particular invocation). For more information on the distinction between system-level and business logic-level faults, the interested reader is referred to [10].

Listing 1 presents a code expert in WSDL for declaring a logical fault that may occur and how it is specified in a BPEL operation process. For more information on these WSDL constructs, the interested reader is referred to [16]. The logical faults declared in web service's WSDL can be encountered upon BPEL process execution at runtime while fault handling mechanisms are taking place if BPEL designer specified one. Web services' developers are strongly encouraged to specify all the logical faults that may occur during web services' operations execution giving to BPEL designer fault handling capabilities in order to resolve this kind of exceptions.

```
<message name="CreditApprovalFaultMsg">
      <part name="approval" element="tns:error" />
</message>
<portType name="CreditApproval">
    <operation name="process">
        <input message="tns:CreditApprovalRequestMsg" />
        <output message= "tns:CreditApprovalResponseMsg"/>
        <fault name="InsufficientCredit"
        message="tns:CreditApprovalFaultMsg" />
    </operation>
</portType>
```

Listing 1. WSDL error type message

3.2 Fault Handling in BPEL

The WS-BPEL 2.0 specification [23] provides fault handling capabilities via the *faultHandler* construct. BPEL programmers are able to deal with logical faults in catch-and-handle fashion. For system faults the WS-BPEL 2.0 specification provides features like "failover" and "retry" to assist developers in dealing with them. Failover strategy determines alternative service invocations, when the first service invocation fails, in contrary to retry strategy that determines a specific time interval between invocation attempts to the same service and the number of invocation retries. A combination of the above system fault handling strategies is presented in Listing 2.

There are, however, other runtime faults that the failover and retry mechanisms cannot handle, for example, if a new service with different interface (other input and output parameters, authentication, etc) has been deployed instead of the one defined in BPEL process. In this kind of fault, the usual strategy adopted by BPEL tools for dealing with is to delegate its handling to a human administrator. Moreover, it is necessary for the BPEL process designer to continuously maintain the BPEL scenarios, keeping the alternate service specifications up-to-date -in case failover strategy is adopted- whenever new such services are introduced or existing ones are withdrawn. In this paper we are proposing a middleware framework to resolve faults raised by system inconsistency, previously named system faults.

```
<properties id="RatingService">
    <property name="wsdlLocation">
        http://localhost:8080/axis/servicesRatingService?wsdl
    </property>
    <property name="location">
        http://localhost:2222/services/axis/RatingService
    </property>
    <property name="retryCount">2</property>
    <property name="retryInterval">60</property>
</properties>
```

Listing 2. BPEL specification for automatic retry and failover

4. THE ASOB FRAMEWORK

The ASOB framework introduces a middleware layer, which acts as a service proxy for web service invocation. As shown in the architectural diagram of Figure 1, the ASOB module operates independently from the BPEL executor, possibly running on a distinct machine. The ASOB module intercepts web service invocations originating from the BPEL executor, places the calls to the actual service providers and arranges for resolving any system exceptions that occur during these invocations. We assume that business processes are comprised by services with a web service interface. The BPEL scenario itself does not need to

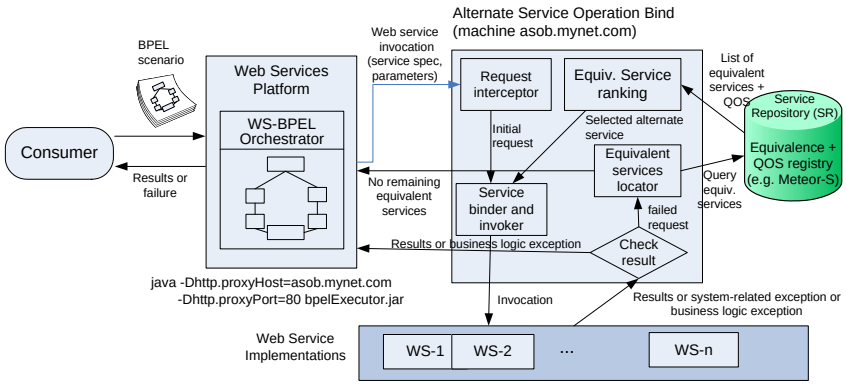

Figure 1. ASOB architecture in a proxy-based setup

include any designer-crafted handlers for system faults, though it may probably include fault-handlers for *application logic-level* faults, which are related to the scenario's business logic and are not handled by the ASOB module. There is also no need to apply preprocessing to the BPEL scenario, as proposed in [10] or used specialized development tools, as described in [8], in order to embed system-generated fault handlers in the BPEL scenario. If the designer has included system fault handlers in the BPEL scenario, these will be activated only if the ASOB module has not managed to resolve the exception, which may occur if no equivalent services are found or if all services in the list of equivalent services have been tried and none of these invocations has succeeded. In order to direct web service invocations to the service proxy (the ASOB module) rather than to the machines actually delivering the services, two techniques may be used. The first one is to designate the ASOB module as a generic HTTP proxy to the module that undertakes the execution of the WS-BPEL scenarios. This is illustrated in Figure 1, where the WS-BPEL orchestrator is Java-based and the Java system properties *http.proxyHost* and *http.proxyPort* are used to specify that all HTTP requests should be directed to port 80 of the machine running the ASOB module. Properties can be set from the Java execution command line, e.g.

```
java -Dhttp.proxyHost=asob.mynet.com -Dhttp.proxyPort=80 \
        bpelExecutor.jar
```

or by adding appropriate coding in BPEL component (JBI) as shown in Listing 3.

In environments where HTTP proxying cannot be designated through system properties, a transparent redirection router may be employed, as illustrated in the architectural diagram of Figure 2. Traffic from the machine executing the WS-BPEL orchestrator is directed to the transparent router (effectively, the transparent router is designated as the default TCP/IP router for the machine executing the WS-BPEL orchestrator), and the router's configuration arranges for forwarding HTTP requests to the ASOB module. Any layer four switch can perform such redirections, while software routing/firewall modules such as *iptables* and *ipf* can be employed as well [17]. In both cases, it is possible to specify that the proxy will not be used if invocations to specific servers are made (e.g. if some services are deployed on a server within the organization's intranet and should be accessed there only). In the first case (Java library proxying) the system property *http.nonproxyHost* can be set to the appropriate value (e.g. *localhost|wsrvs.myCorp.com*) while in the second case (transparent redirection router) rules dictating that traffic to the specific hosts is routed normally should precede the generic traffic redirection rule in the router's configuration.

```
Properties systemSettings = System.getProperties();
systemSettings.put("http.proxyHost", "asob.mynet.com");
systemSettings.put("http.proxyPort", "80");
systemSettings.put("http.nonproxyHost", "localhost|wsrvs.myCorp.com");
systemSettings.put("http.agent", "supplementary options");
```

Listing 3. Applying proxy at programming level

The components internally comprising the ASOB module and the overall ASOB framework operation are described in the following sub-sections.

4.1 ASOB Components and Functionality

The ASOB module consists of five components, discussed in the

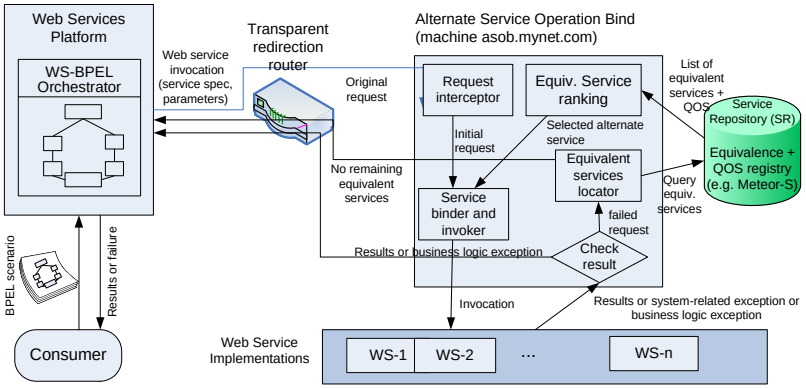

Figure 2. ASOB architecture in a redirection router setup

following paragraphs.

- the Request Interceptor (RI) component. RI intercepts the web service call. In order to be able to accept any request, the request interceptor is not a web service itself (which would require it to adhere to some specific WSDL) and thus it does not run in a web service container; instead, it is crafted as a Java Server Page (JSP), and the JSP container is instructed to run this page upon every request. RI extracts the original web service specification and the payload (i.e. the SOAP-encoded message containing the parameters to the request), and passes them to Service Binder and Invoker module.
- the Service Binder and Invoker component (SBI). SBI is responsible for invoking a specific web service. It accepts a web service specification and a payload, and arranges for invoking the particular web service, attaching the payload to the invocation.
- the Equivalent Service Locator module (ESL). ESL is the component which discovers functionally equivalent web services to a designated one (in the context of ASOB operation, this is always the service specified in the original request). This is performed by querying an appropriate repository.
- the Service Repository (SR). SR is a repository containing up-to-date web services specification entries (WSDL location, Endpoint addresses, Operation interfaces). In addition to this information, SR should *at least* provide means for identifying functionally equivalent groups of services using semantic tagging (as, for example METEOR-S and WSMX) or any other suitable approach. If the repository, additionally to functional equivalence, can provide information on the Quality of Service (QoS) characteristics of the services, ASOB can exploit this information to resolve exceptions more efficiently.
- the Equivalent Service Ranking component (ESR). ESR module is responsible for sorting the equivalent service list according to QoS criteria, in order to make the exception resolution process more efficient. In order to perform this sorting, the ESR component exploits data collected from previous web service invocations, and more specifically service availability and service response time. If SR contains additional QoS attributes, these can be exploited as well. In order to simplify our discussion in the remaining of this paper, we will assume that SR contains data on service availability and service response time, and the SBI component updates this data when each web service invocation completes, either successfully or with a failure.

In the following sub-section we describe in detail how a web service invocation is performed within the ASOB framework and how exceptions are resolved.

4.2 Request Processing in the ASOB framework

Consider a BPEL scenario that has been designed and deployed in the web services platform of figure 1, where an appropriate WS-BPEL orchestrator resides. At some time point, the execution of the BPEL scenario commences and a web service invocation is executed. At this stage, either the execution environment sends the request to the ASOB module, if the latter has been designated as an HTTP proxy to the former [Figure 1], or the network packets comprising the request will be sent to the transparent redirection router, which will forward them to the ASOB module [Figure 2]. In both cases, the request will reach the ASOB module, where it will be intercepted by the RI component.

The ASOB RI component inspects the web service call request and extracts from it the web service specification (i.e. the requested URL) and the payload (the XML document containing the SOAP envelope) from it, and passes these chunks to SBI. SBI invokes the web service –at this stage, this is the service originally specified by the BPEL designer- and waits for the response. At this stage, one of the following possibilities may occur:

1. the service does not return a reply within a predefined amount of time. This is typically owing to the host providing the service being down or unreachable. SBI informs SR that the service has been found to be unavailable (thus SR updates the service's availability qualitative characteristic) and passes control to the ESL module, to initiate the exception resolution.
2. an exception of type "no route to host" or "connection refused" is returned, then the service is again unavailable and the same actions as in the previous case are taken.
3. a network-level exception of type "unknown host" is returned. This exception indicates that the host that offered the service has ceased to exist, thus the service has become permanently unavailable. In this case, the SR is notified that the service should not be considered any more as a candidate substitute for other failed services which were functionally equivalent to its original specifications. Subsequently, control is again passed to the ESL module for resolving the exception.

   Note that the failed service is *not* withdrawn from SR, because such a withdrawal would break the process of finding alternatives to the specific service (i.e. if a WS-BPEL scenario contained an invocation to that service, then the registry would not contain any information on which other services are functionally equivalent to it, so as to enable the ASOB module to invoke an alternative implementation). SR may periodically check whether some "blacklisted" service has become again available, and remove the "blacklist" flag, possibly reducing in parallel the service's availability and reliability QoS characteristics.
4. a reply is received from the web service container (as opposed to the previous cases where no reply is received). In this case, the reply is checked and one of the following actions are taken:
   a) if the reply is a "Unknown service" or "Parameter mismatch" fault (these fault types are produced by the web service container), either the service has been withdrawn from the host ("Unknown service") or the service's interface has been modified and is now incompatible to the specifications expected by the application. These fault types correspond to permanent errors, thus processing continues as in case (3) above.
   b) if the reply is a "normal" response (i.e. it is a valid *output message* declared in the service's WSDL), the reply is returned to the WS-BPEL script that made the invocation;

the time taken for the service to reply is noted, SR is notified of the service's availability and response time, to update the respective QoS characteristics accordingly and request processing concludes.

c) if the reply is not a valid *output message*, it corresponds to a fault. In this case, ASOB attempts to discriminate between application logic faults and other system-oriented faults. This is accomplished by checking whether the reported fault is declared in the web service's WSDL description as a fault type or not; if it is, the reply constitutes an application-level exception which must be passed back to the WS-BPEL script, where it will be probably caught and handled via an appropriate *Catch* construct; therefore, this reply is handled as a normal reply [case (b), above], i.e. it is returned to the WS-BPEL script, SR is notified of the service's availability and response time, to update the respective QoS characteristics accordingly and request processing concludes.

   If the reply is neither a valid *output message* nor a declared fault, then it is considered to be a system-level error due to service unavailability or temporary malfunction (e.g. inability to access a local database); this case is handled similarly to other transient errors, i.e. cases (1) and (2).

This stage of processing is reached if the original web service invocation has failed due to a system error. ESL locates equivalent web service operations, by issuing a query to SR. The internals of query processing by SR are outside the scope of this paper, and any pertinent technology (e.g. [12], [13] and [14]) can be used. The query to SR will contain the endpoint URL of the operation that failed (e.g. http://www.domain.com/WebService/WSOperation), and SR's reply will include a list of records, with each record corresponding to an equivalent service that can be substituted for the failed one. Each such record will contain at least the service's endpoint and target namespace[1] , while the service's response time, availability and other QoS characteristics may be included, if available in SR.

Having received the alternative services list, the ESR component checks whether QoS attributes are included in it, and in particular *response time*. If this attribute is included, ESR sorts the list in ascending order of this attribute, thus services with smaller response time are placed first. The sorting criteria can be modified by the ASOB administrator to consider alternate attributes (e.g. availability, throughput and so forth [18]) or attribute value combinations (e.g. [(70% / response_time) + 30% * availability)]. If no QoS attributes are included in SR's reply, the sorting step is skipped and the elements remain in the order they were presented by SR.

Finally, ESR iterates over the alternate services list, starting from the first (if sorting has taken place, the "most preferred" one will be placed there) and moving towards the last ("least preferred"). For each such service, the namespace in the payload is adjusted and the web service is invoked; the results of each invocation are handled as described in steps (1)-(4) above, except for that in case of a failure, where the next service in the list is tried rather than invoking the ESL component anew. This stage can be repeatedly performed until an alternate functionally equivalent service responds as expected or until a number of attempts (defined by the ASOB administrator) has been reached. If the maximum number of unsuccessful attempts is reached or the list is exhausted (or if it was initially empty), resolution is possible for the exception and a special type of fault, namely PolicyFault, is returned to the BPEL execution environment. The BPEL designer may have included an appropriate fault handler in the BPEL script, which will attempt to follow an alternate business process for remedying the fault.

Listings 4 and 5 illustrate the way requests are processed in ASOB using pseudo-code. One issue that must be noted here is that the Service Repository update is an asynchronous task. Currently, it is initiated as a separate thread immediately before results are returned to the WS-BPEL script that placed the original invocation; an alternative, more efficient approach would construct update batches and submit them to the registry when a certain amount of updates have been amassed or at specific time intervals.

```
/* Variables list */
ewsLIST /* list of equivalent web service */
OWS /*originally selected web service */
owsRS /*web service operation invocation results of OWS */
EWS /*equivalent web service to the originally selected */
ewsRS /*web service operation invocation results of EWS */
finRS /* final result */
knownEXC /* exception listed in OWSs WSDL */
policyFAULT /* fault when equivalent services has been exhausted
    and exception resolution was not achieved */

OWS ← getPart(request, "webService");
payload ← getPart(request, "payload");
owsRS ← invoke_bind_WS(OWS, payload);
if (not timeout_occured)
    if (is_normal_reply(owsRS))
        finRS ← owsRS;
    else
        /* Initialize list of known exceptions */
        knownEXC ← getExceptionsFromWSDL(OWS);
        if (owsRS in knownEXC)
            /* Known, application-logic exception, return it */
            finRS ← owsRS;
        else
        /* Unknown, system exception, attempt resolution */
            ewsLIST ← getEquivServices(OWS);
            ewsRS ← invoke_alternate_ws(ewsLIST, payload);
            finRS ← ewsRS;
        end if
    end if
end if
return finRS to BPEL execution environment;
exit();
```

Listing 4. Invocation algorithm

5. ASOB Implementation and Performance Analysis

In order to assess the performance impact of the exception resolution scheme presented in section 4, we implemented ASOB and performed extensive tests using various configurations. Our

[1] The namespace is required for technical reasons, since the payload complementing a web service invocation should refer to the namespace declared by the web service in its WSDL file.

main goal was to evaluate the overhead incurred due to the introduction of the additional middleware layer, both in terms of request processing time and request throughput, and to gain insights on the solution's scalability in terms of concurrent invocation handling. In the following sub-sections we describe the test environment and the obtained results.

```
begin function invoke_alternate_ws (in ewsLIST, in payload, out ewsRS)
    ewsLIST ← sort ewsLIST according to ASOB-wide criteria
    while ewsLIST not empty
        EWS ← first_element(ewsLIST)
        ewsRS ← invoke_bind_WS(EWS, payload)
        if (not timeout_occured)
            if (ewsRS in knownEXC)
            /* Known application-logic exception, return it */
                return ewsRS;
            else if (is_normal_reply(ewsRS))
            /* No exception occurred, return reply */
                return ewsRS;
            end if
        end if
        /* This code is reached if a timeout occurred or
        an unknown exception (system exception)
        was thrown. Proceed with next alternate service */
        ewsLIST ← remove first_element(ewsLIST);
    end while
    /* All alternate services have been tried and have failed */
    return policyFAULT;
end function

function invoke_bind_WS(in WS, in payload, out result)
    performBinding(WS);
    result ← invoke(WS, payload);
    if (timeout_occured)
        update_SR_QoS(WS, NULL, Unavailable, Transient);
    else if (is_normal_reply(result) || result in knownEXC)
        update_SR_QoS(WS, responseTime, Available, NULL);
    else if ((result == UnknownSrv) || (result == ParamMismatch))
        /* service has been withdrawn or updated to an incompatible and
            thus unusable form */
        update_SR_QoS(WS, NULL, Unavailable, Permanent);
    else
        update_SR_QoS(WS, NULL, Unavailable, Transient);
    end if
end function

function update_SR_QoS(in WS, in respTime, in isAvail, in errType)
    /* start a new thread to forward QoS information to SR; return
immediately */
end function
```

Listing 5. Invocation routines

5.1 Testbed configuration

For our experiment we used three distinct machines: the first machine (AMD Athlon XP 2000+ processor, 1GB of RAM, Windows 2000 Server) executed the BPEL scenarios, which were developed using Netbeans's 6.0 IDE BPEL designer solution [19] and executed using the WS-BPEL 2.0 compliant Java Business Integration (JBI) component [20], deployed on Glassfish v2 application server [21]. The second machine (Pentium 4, 3GHz processor, 1GB of RAM, Windows 2000 Server) was hosting the ASOB module (operating within a Glassfish v2 application server) and the service repository, implemented as a MySQL database. The third machine(Pentium 4, 3GHz processor, 1GB of RAM, Windows 2000 Server) hosted the target web services. Both the proxy and the transparent router architectures were considered having almost identical results, thus in the remaining of this section we will only present data collected for the proxy setup. For benchmark data collection we used the benchmarking tool from Apache, *ab* [22] and internal timers. In all the measurements and diagrams presented below, all the originally selected web services were available and no system fault occurred, and we forced however ASOB to retrieve and sort the list of equivalent services. This approach was chosen because service unavailabilities are reported with large variances in time (depending on the root cause – e.g. the expiration of a timeout is detected after much longer time as compared to a "no route to host" error), and such a behavior would not allow results to be conclusive.

Note that the obtained results actually depict a worst case for ASOB, since ASOB performs exception resolution operations (i.e. retrieval and sorting of the alternate services list - of course alternate services are not invoked), while the non-mediated invocations are not penalized in any way. Note also that in the absence of ASOB, the BPEL processes would have crashed anyway in the presence of errors.

5.2 Performance Analysis

Figure 3 illustrates the ASOB internal process time against the database size (denoted as *db-size*) and the number of equivalent services identified in the repository (denoted as *qws* in the diagram legends). ASOB's internal process time PT_{asob} does not include the actual web service invocation time or the time needed for the extra network message transmission (from the BPEL script to the middleware – it does however include the time to send back the reply, since this activity is ASOB-initiated and can be thus measured using internal timers) and comprises of the following four terms:

- t_1: web service call interception duration
- t_2: equivalent web service discovery duration
- t_3: equivalent service ranking and sorting duration
- t_4: time to send back the results of actual web service invocation

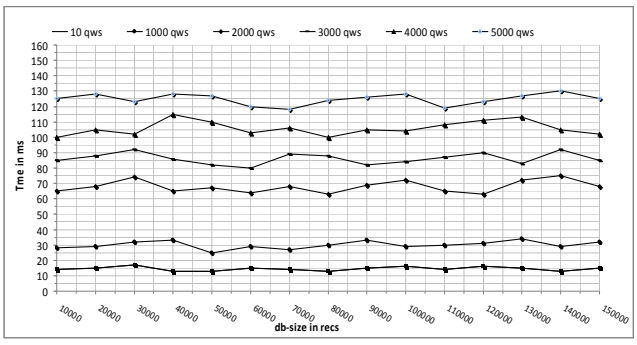

Figure 3. ASOB internal process time

Figure 3 indicates that the internal process time (y-axis) mainly depends on the number of equivalent services found in the services' repository (SR component) and not on the SR's size (x-axis). The very small dependency on the overall SR database size

can be attributed to the use of appropriate indexes within the database, which effectively exclude the non-relevant tuples from the database's search, thus only 1-5 extra disk page accesses are performed. The overhead increment, on the other hand, when the number of alternate services increases is considerable, mainly affecting terms t_2 (due to query process time by the database and interprocess communication/process switching to transfer the result into the ASOB module) and t_3 (sorting of the result, typically of complexity *O(n * log(n))*. In real-world cases, however, it is expected that the number of equivalent services will be at most a few tens, thus the overhead introduced per invocation will be in the range of 15-25 msec, which is typically a very small fraction of the overall web service processing time.

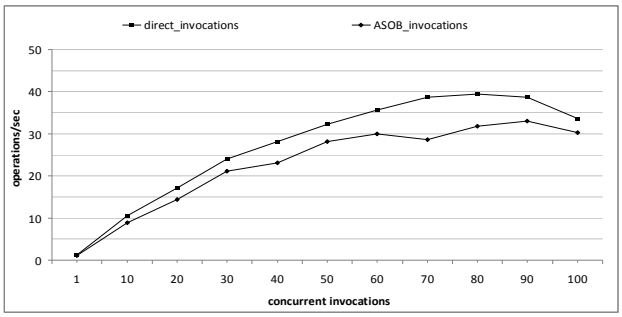

Figure 4. ASOB-mediated vs. direct WS invocation throughput

Figure 4 illustrates the number of invocations that can be served in a unit of time against the number of concurrent invocations when (a) services are directly invoked and (b) when invocations are made through the ASOB middleware. The latter case exhibits a throughput drop of 8-16%, except for the case of 70 concurrent invocations, where the drop is approximately 25%. The throughput drop is to be expected, since the use of ASOB incurs the overhead of "extra processing" by the ASOB on top of the processing required by the direct invocations. The anomaly exhibited at the area of 70 ASOB-mediated concurrent invocations can be attributed to various systemic reasons, such as garbage collection, thread thrashing, buffer flushing, etc, separately or to a combination of those reasons. From the above diagram, it is also evident that at the point of 80 concurrent direct invocations –that is without ASOB middleware– the maximum throughput is reached. Diminishing rate of concurrent invocations per second is apparent when further increasing the number of invocations; at the point of 100 direct invocations the drop in performance is so sharp (the machine resource's limits have been reached) that there is no point in considering more concurrent invocations.

For ASOB-mediated invocations it seems that the peak is reached at the area of 90 concurrent invocations, while diminishing rates are observed at the area of 100 concurrent invocations. In the area between 80 and 100 concurrent invocations, the difference between direct and ASOB-mediated invocations is gradually becoming smaller. This phenomenon can be attributed to the fact that while the web service execution machine has reached its limits regarding request processing at 80 direct invocations, there is still ample power available in the machine hosting the ASOB module to handle the processing required by the specific module. Summarizing the difference between the curves, we can roughly observe three areas in the diagram:

1. from 1 to 50 aINV, where the difference between the two is small or slightly increases (8%-12%)
2. from 50 to 80 aINV, where the difference between the two is higher (12%-16%)
3. from 80 to 100 aINV, where the difference is diminishing and tends to return to the difference observed in the first area, which however is owing to the diminishing direct invocation performance and not to increased performance of ASOB.

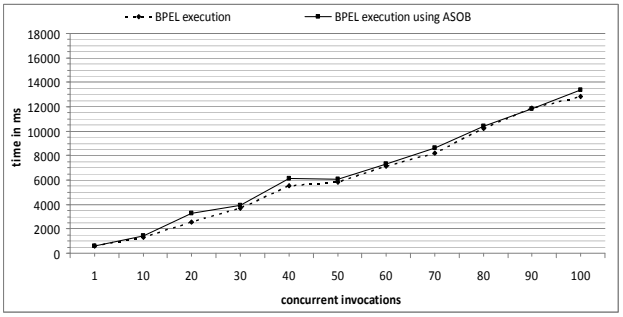

Figure 5. BPEL scenario execution time

Figure 5 illustrates the BPEL execution process time of a certain BPEL scenario against the number of concurrent invocations. The BPEL scenario consisted of three web services –all accessible at the same server-, whose execution time was 100msec, 100msec and 300msec. The x-axis represents the concurrent BPEL process executions (or invocations if BPEL process provides a web service interface) and y-axis the time passed until all of them completed successfully. It is obvious that the BPEL process time is slightly increased in the ASOB-mediated case, but the increment is very small – less than 6% in all cases, except for two anomaly points (20 and 40 concurrent invocations) where an overhead of 23% and 14% is observed, owing probably to reasons as those listed for Figure 4 above.

Figure 6 depicts the BPEL scenario execution throughput against the number of concurrent invocations. The behavior is consistent with the previous diagrams, i.e. the throughput drop is in the range of 6%-10% in the normal cases, whereas two anomaly points (20 and 50 concurrent invocations) are observed, for which performance drops are quantified to 20% and 24%. Again, reasons as those listed for Figure 4 above may explain these anomalies. The peak throughput in this diagram appears to be reached in the range of 50-70 concurrent invocations, and beyond that point performance starts to drop.

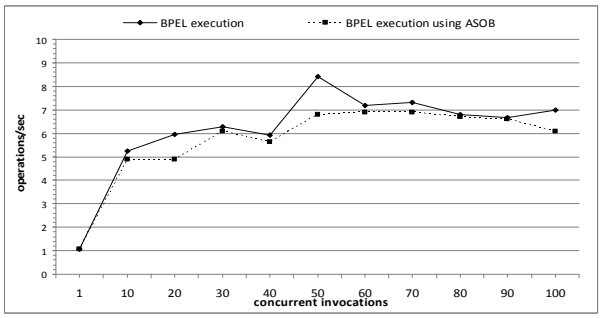

Figure 6. ASOB-mediated vs. direct invocation BPEL scenario execution throughput

6. Conclusions and future work

Using BPEL to model business processes has many advantages, including adherence to standards and speed of development and deployment; due to the distributed nature of the target environment, however, exceptions often arise during the BPEL scenario executions and the exception handling mechanisms provided by BPEL are too rigid to flexibly adopt to the continuous SOA environment updates and to the diversity of the exception causes. The work presented in this paper caters for the resolution of exceptions generated due to system faults, such as host unavailabilities or network errors, relieving thus the WS-BPEL scenario designer from the burden of specifying (and updating) handlers for these fault types and restricting exception handling in the WS-BPEL scenario to the application-logic faults only. The proposed approach uses a middleware layer, which exploits a repository of functionally equivalent services and attempts to remedy system faults by invoking a service equivalent to the failed one. The middleware also issues updates to the repository, notifying it of the services' observed availabilities and response times, and these QoS characteristics are taken into account when a replacement service needs to be selected for substituting a failed one.

One direction for further research is to consider different criteria for replacement service selection *for each invocation*, rather than specifying these criteria on an installation basis. To this end, each invocation will need to be complemented with a specification of these criteria, and mechanisms to this end must be devised. The exploitation of the *http.agent* standard system property could be considered for hosting this criteria, but since its value typically remains constant throughout the execution of a BPEL scenario, more fine-grained mechanisms are called for. Another envisaged extension is the intervention of ASOB so as to modify even the original invocations specified by the WS-BPEL scenario designer, taking into account QoS criteria, selecting thus the optimal service for each task and not necessarily the originally specified one.

7. REFERENCES

[1] Leymann, F., Roller, D., and Schmidt, M.-T. 2002. Web services and business process management, IBM Systems Journal, Vol. 41, 198 No2.

[2] Newcomer, E. and Lomow, G., 2005. Understanding SOA with Web Services, Addison-Wesley.

[3] Dellarocas, C. and Klein, M., 2000. A knowledge-based approach for handling exceptions in business processes, Information Technology and Management, 1, 3 (2000) 155-169.

[4] ActiveVOS , 2008. Active Endpoints presentation, http://www.activevos.com/index.php

[5] Oracle Corporation, 2008. Oracle BPEL Process Manager, http://www.oracle.com/technology/bpel/

[6] The Eclipse BPEL Team, 2008. The Eclipse BPEL Project, http://www.eclipse.org/bpel/

[7] Java.Net, 2008. Open ESB, https://open-esb.dev.java.net/

[8] Dobson, G., 2006. Using WS-BPEL to Implement Software Fault Tolerance for Web Services. Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA'06).

[9] Kareliotis C., Vassilakis C., Georgiadis P., 2006. Towards Dynamic, Relevance-Driven Exception Resolution in Composite Web Services, 4th International Workshop on SOA & Web Services Best Practices, Portland, Oregon, USA at OOPSLA.

[10] Kareliotis C., Vassilakis C., Georgiadis P., 2007. Enhancing BPEL scenarios with Dynamic Relevance-Based Exception Handling, Proceedings o f the IEEE 2007 International Conference on Web Services (ICWS).

[11] CA Willy Technology, 2007. SOA and Web Services – The Performance Paradox, http://www.ca.com/us/whitepapers/collateral.aspx?cid=147947

[12] Kochut, K. J., 1999. METEOR Model version 3. Athens, GA, Large Scale Distributed Information Systems Lab, Department of Computer Science, University of Georgia.

[13] Verma, K., Sivashanmugam, K., Sheth, A., Patil, A., Oundhakar, S., Miller, J., 2005. METEOR-S WSDI: A Scalable Infrastructure of Registries for Semantic Publication and Discovery of Web services. Journal of Information Technology and Management, Special Issue on Universal Global Integration, 6, 1 (2005) 17-39

[14] Cimpian, E., Moran, M., Oren, E., Vitvar, T., Zaremba, M., 2005. Overview and Scope of WSMX. Technical report, WSMX Working Draft, http://www.wsmo.org/TR/d13/d13.0/v0.2/

[15] Feier, C., Roman, D., Polleres, A. Domingue, J., Stollberg, M., Fensel, D. (2005). Towards Intelligent Web Services: Web Service Modeling Ontology, In Proc. of the International Conf on Intelligent Computing (2005)

[16] Angelov D. et al., 2007. WSDL 1.1 Binding Extension for SOAP 1.2, http://www.w3.org/Submission/wsdl11soap12/#faultelement

[17] Wessels, D., 2001. Interception Proxying and Caching, in Web Caching, O'Reilly, ISBN: 1-56592-536-X.

[18] Al-Masri, E., 2008. The QWS Dataset, http://www.uoguelph.ca/~qmahmoud/qws/index.html

[19] NetBeans Project, 2008. Netbeans IDE http://www.netbeans.org/

[20] JBI Team, 2008. Java Business Integration, https://open-esb.dev.java.net/Components.html

[21] Glassfish Team, 2008. Glassfish Open Source Application Server https://glassfish.dev.java.net/

[22] Apache foundation, 2007. ab Apache Open Source benchmarking tool. http://httpd.apache.org/docs/2.0/programs/ab.html

[23] OASIS, 2007. OASIS Web Services Business Process Execution Language (WSBPEL) TC. http://www.oasis-open.org/committees/wsbpel/