QoS-aware Exception Resolution for BPEL Processes: A Middleware-based Framework and Performance Evaluation

Kareliotis Christos
Ph.D. Candidate
Dept. of Informatics and Telecommunications, University of Athens, Greece
+302107275220
ckar@di.uoa.gr

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

Efstathios Rouvas
Ph.D. Candidate
Dept. of Informatics and Telecommunications, University of Athens, Greece
+302107275220
rouvas@di.uoa.gr

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

ABSTRACT

WS-BPEL is widely used nowadays for specifying and executing composite business processes within the Service Oriented Architecture (SOA). During the execution however, of such business processes, a number of faults stemming from the nature of SOA (e.g. network or server failures) may occur. The WS-BPEL scenario designer must therefore use the provisions offered by WS-BPEL to catch these exceptions and resolve them, usually by invoking some equivalent web service that is expected to be reachable and available. System fault handler specification is though an additional task for the WS scenario designer, while the presence of such handlers within the scenario necessitates extra maintenance activities, as new alternate services emerge or some of the specified ones are withdrawn.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 process of selecting the alternate services to be invoked can be driven by process consumer-specified QoS policy, specifying lower and upper bounds for each QoS attribute as well as the importance of each QoS parameter. Moreover, the middleware arranges for bridging syntactic differences between the originally invoked services and functionally equivalent replacements to it, by employing XSLT-based transformations. The middleware is deployed and maintained independently of the WS-BPEL scenarios, removing thus the need for specifying and maintaining system fault handlers 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, in order 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 within WS-BPEL scenario executions and resolves them by invoking operational replacement services that are functionally equivalent to the failed ones. ASOB acts as a web service proxy, so it intercepts and arranges for processing service operation invocations performed during the execution of WS-BPEL scenarios. The real invocation is thus performed by ASOB, and therefore ASOB is directly notified of 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 process of selecting the alternate services to be invoked can be driven by process consumer-specified QoS policy, specifying lower and upper bounds for each QoS attribute as well as the importance of each QoS parameter. Moreover, the middleware arranges for bridging syntactic differences between the originally invoked services and functionally equivalent replacements to it, by employing XSLT-based transformations.

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 regarding fault handling. Section 4 discusses service equivalence and service QoS characteristics, while section 5 presents the overall architecture of ASOB and gives details on the functionality of its modules. In section 6 we discuss design and implementation considerations, while section 7 presents performance results obtained by extended benchmarking of ASOB-mediated BPEL process execution. Finally in section 8 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 Orchestra [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.

[12] and [13] present AgFlow, which revises the execution plan in order to conform the user's QoS constraints. AgFlow may operate either using *global planning*, in which the execution plan is revised to meet the QoS constraints specified by the user, or using *local optimization*, in which optimization is made on individual task basis, using the Simple Additive Weighting [14] technique to select the optimal service for a given task. [15] presents VieDAME, which performs BPEL scenario adaptation on the basis of QoS parameters, but these QoS parameters and the selection strategy are pre-determined through pluggable modules; moreover, VieDAME is implemented using extensions available only in the ActiveBPEL engine [16], and is thus platform-dependent. [17] introduces end-user specified policies through QoSL4BP, and BPEL transformers that incorporate the policies and appropriate monitors to the BPEL scenario before its execution. This work mainly targets at monitoring the execution and raising exceptions when the desired QoS are not met, rather than adapting the BPEL scenario so as to best match the QoS demands of the scenario consumer. [18] introduces another BPEL extension and uses an extended BPEL engine to deliver QoS-based adaptation; the use however of a BPEL extension and a custom execution engine are potential barriers to the adaptation of this solution.

[19], [20] and [21] consider service BPEL scenario adaptation in the context of exception resolution. [19] creates exception-aware process schemas, and the infrastructure detects invocation faults and substitutes services that have failed with alternate ones; QoS characteristics are not considered in this work. [20] includes QoS characteristics in the alternate service replacement, it does not allow however the specification of the replacement policy by the process consumer. [21] uses autonomic computing concepts for providing execution plan formulation for business processes, taking into account QoS parameters, monitors dynamically QoS violations at runtime and provides instrumentation for the handling of these exceptions, while [22] performs composite service re-planning during composite service workflows execution and [23] uses parallel execution of BPEL documents using user defined QoS.

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 [24], [25] in cooperation with WSMX (Web Services Execution Environment) [26]. 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 [27]. 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 [28]. 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 a more detailed discussion on the distinction between system-level and business logic-level faults, the interested reader is referred to [10].

Listing 1presents 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 [29]. The logical faults declared in web service's WSDL can be thrown upon BPEL process execution at runtime, while fault handling mechanisms can be specified by the BPEL designer to intercept and react to these faults. Web services developers are strongly encouraged to specify all the logical faults that may occur during web services' operations execution, allowing thus the BPEL designer to anticipate specific faults and suitably employ fault handling capabilities to resolve this kind of exceptions.

3.2 Fault Handling in BPEL

The WS-BPEL 2.0 specification [30] 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. An example of the *retry* fault handling strategy is presented in Listing 2:

```
<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

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

4. SERVICE EQUIVALENCE AND QoS CHARACTERISTICS

Two fundamental issues that must be addressed to enable exception resolution through invocation of some alternate implementation are the following:

1. how to determine which services are equivalent to the failed one and can be thus used as substitutes
2. among all equivalent services identified, which can be considered as the "best" choice in the context of a particular invocation.

These two issues will be discussed in the following paragraphs.

4.1 Service Equivalence

According to [31], "semantic equivalent Web services are a group of Web services, which can replace each other, if a Web service is invalid at runtime of a composite service, another semantic equivalent Web service can be selected to replace it and make the composite Web service running on". [15] further elaborates on the definition of equivalent services, identifying two major equivalence types, namely *syntactic equivalence* and *semantic equivalence*. Syntactic equivalence indicates that the services perform the same function *and* the interfaces of the original and the alternative services match, as is typically the case when multiple instances of the same service are hosted on different machines to provide increased reliability or load balancing. Semantic equivalence, on the other hand, indicates that the services only have the same functionality but expose it using different interfaces; this mainly occurs when considering services coming from completely different providers on the Web.

In this paper, we will adopt the approach presented in [15] for service equivalence. We must note here that, while it is quite straightforward to substitute some service invocation with an invocation to a *syntactically equivalent* one (only the endpoint needs to be changed), using a *semantically equivalent* service as a replacement necessitates the introduction of steps to align the payload created by the client (which matches the interface of the originally invoked service) to the interface exposed by the replacement service. For instance, consider the services *PersonLocator* and *PersonLocator2,* accepting payloads as depicted in Listing 3 and Listing 4 respectively. An invocation to *PersonLocator* can be substituted by an invocation to *PersonLocator2,* provided that prior to the invocation of *PersonLocator2* the original, client-generated payload is appropriately transformed. Note that this transformation may include additional items not depicted in this figure, such as the namespace. [15] addresses this issue by employing XSLT-based transformations. For the moment, we will assume that for every pair of semantically equivalent services S_1 and S_2, four XSLT stylesheets are available, namely *PayloadS1toS2, ReplyS1toS2, PayloadS21toS1 and ReplyS2toS1*, affording for the transformation of both the payloads and the respective replies among the service interfaces. Under this provision, the substitution of S_1 by S_2 is handled as illustrated in Figure 1.

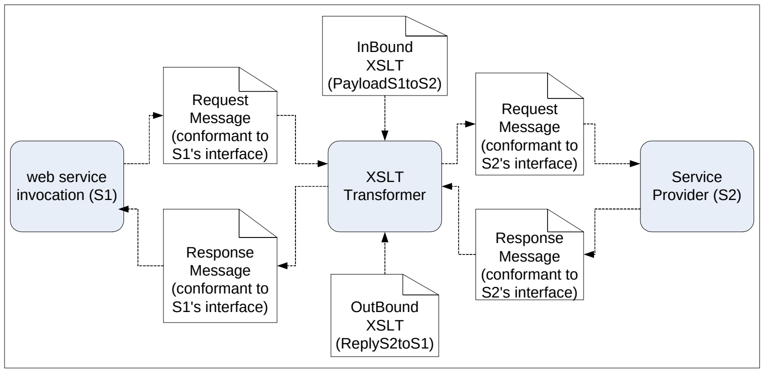
We will revisit the issue of XSLT transformations in section 6.1, to discuss a tradeoff between XSLT stylesheet repository manageability and performance.

```
<PersonalInfo>
  <Name>John</Name>
  <Surname>Smith</Surname>
  <Country>USA</Country>
</PersonalInfo>
```

Listing 3. Payload for a *PersonLocator* service

```
<PersonDetails>
  <FullName>
    <FirstName>John</FirstName>
    <LastName>Smith</LastName>
  </FullName>
  <Country>USA</Country>
</PersonDetails>
```

Listing 4. Payload for a semantically equivalent (but syntactically different) PersonLocator2 service



InBound XSLT (PayloadS1toS2)
Request Message (conformant to S1's interface)
Request Message (conformant to S2's interface)
web service invocation (S1)
XSLT Transformer
Service Provider (S2)
Response Message (conformant to S1's interface)
Response Message (conformant to S2's interface)
OutBound XSLT (ReplyS2toS1)

Figure 1. ASOB architecture in a proxy-based setup

4.2 Quality of Service Characteristics

A service, besides its functionality, is characterized by a set of qualitative attributes that describe various non-functional, yet important, aspects of its real-world behaviour. The set of these attributes is commonly referred to as *QoS attributes* or *QoS characteristics* and typically include aspects such as response time, security, cost, availability and so forth. In many cases, it is important to consider the QoS attributes of a service and its candidate replacements, since using a replacement service with very different QoS characteristics than the originally specified one may have undesirable side-effects, e.g. using a service with much higher cost will lead to excessive charging of the service consumer, whereas using a bank transaction service with significantly lower security may cause the leaking of credit card numbers. In the following paragraphs we will briefly discuss the QoS aspects considered in the ASOB framework and the mechanisms for specifying QoS-related constraints and criteria, to be considered in the selection of replacement services.

4.2.1 QoS Attributes

Many works insofar have discussed issues related to the non-functional aspects that can be represented through QoS attributes, how such representations can be organized (e.g. through taxonomies) or how non-functional aspects can be quantified ([32], [33], [34], [35], [36], [37], [38]). The most typical QoS attributes reported in the literature are listed below; a more detailed discussion on these attributes is beyond the scope of this paper, and the interested reader is referred to the citations given above.

- **Cost,** q_c, representing the cost-related and charging-related properties of a service [36].
- **Reputation,** q_{re}, reflecting a measure of its trustworthiness, mainly dependent on end users' experiences of using the service
- **Successful Execution Rate,** q_{suc}, expressing the probability that a request to the particular service is successfully concluded within the maximum expected time frame indicated in the Web service description.
- **Availability**, q_{av}**,** of a service s is the probability that the service is accessible to be invoked.
- **Accuracy,** q_{acc}, reflecting the error rate generated by the service.
- **Performance,** q_{perf}, expressing how fast a service request can be completed. Often, this is a composite attribute including elements such as *response time*, *throughput* etc.
- **Security**, q_{sec}, reflecting the ability of a service to provide identification, message authentication, confidentiality, non-repudiation and other security-related functionalities.

- **Reliability**, q_{rel}, expressing the ability of a service to perform its functions (to maintain its service quality).
- **Robustness**, q_{rob}, showing the ability of the service to function correctly in the presence of incomplete or invalid inputs.
- **Scalability**, q_{scal}, representing ability of the service to process more requests in a certain time interval.
- **Fidelity**, $q_{fed,}$ used to determine how well services are meeting expected user requirements [38].
- **Network-related QoS,** q_{net}, w accounts for the QoS mechanisms operating in the underlying network which are independent of the service.

For brevity, and without loss of generality, in this paper we will consider in our discussion only a subset of these QoS attributes, namely *cost*, *response time*, *availability*, *security* and *throughput*. For notational convenience, we will refer to these QoS attributes as q_1, q_2, …, q_5, following the order they are listed above.

Regarding the units and scale used for measuring the qualitative attributes, again without loss of generality, we will consider that each QoS attribute is quantified as an integer in the range 1 to 5. Using such a normalized range is a common approach in repositories hosting QoS attributes, particularly when multi-attribute criteria need to be supported (e.g. [32], [39], [40], [41], [42]).

Table 1 presents an example of QoS attribute value normalization, where values of different scales and units are mapped into the range [1, 5]. This normalization is typically performed by the repository operator.

	QoS provider 1	**QoS provider 2**	**value**
q_1: Cost	10 €	11 €	1
q_2: Response time	0.0001 ms	Real-time	5
q_3: Availability	High	> 95%	4
q_4: Security	DES/3DES	128 bits	3
q_5: Throughput	High throughput	99%	5

Table 1. Normalizing the unit and scale

4.2.2 Quality Vectors

In order to enable the selection of the “most suitable” replacement service when some invocation fails, means for expressing the QoS requirements for candidate services and for quantifying unambiguously the overall “suitability” should be afforded.

In our approach, for each invocation, we consider three vectors that comprise the *service replacement policy P* for this particular invocation. The policy P is thus a triple *(MAX, MIN, W)*, where *MAX*, *MIN* and *W* are quality vectors (defined below). The first and the second include the upper and the lower bounds –respectively- of a qualitative attribute the user wants to be satisfied; effectively these two vectors which comprise the *quality constraints*. The third vector represents the qualitative attribute’s corresponding weights, i.e. how important each qualitative attribute is considered by the consumer in the context of the particular execution. Higher weights (in absolute value) indicate higher importance of the specific qualitative attribute. Thus, considering the five QoS attributes listed above, the quality vectors MAX, MIN and W can be defined as:

$MAX=(max_{q1}, max_{q2}, max_{q3}, max_{q4}, max_{q5})$

$MIN=(min_{q1}, min_{q2}, min_{q3}, min_{q4}, min_{q5})$

$W=(w_{q1}, w_{q2}, w_{q3}, w_{q4}, w_{q5})$

Null value specifications are allowed in W_i, in which case they are substituted by zeroes. Specification of negative values for specific elements of *W* may be employed by the invoker to designate that services having smaller values for the specific qualitative attributes are preferred against those having higher values; cost and response time are examples of qualitative attributes for which negative values are expected to be used. Finally, the ASOB framework administrator defines a *default weight vector* for the cases that all elements of W are zero. To exemplify the above, the policy specification

MIN(0, 0, 3, 4, 0), MAX(2, 0, 0, 5, 0), W=(-0.4, 0.1, 0.2, 0, 0)

indicates that we only services having q_1 (cost) ≤ 2, q_3 (availability) ≥ 3 and 4 ≤ q_4 (security) ≤ 5 should be considered, while for QoS attributes q_2 (response time) and q_5 (throughput) no restriction is placed. At the subsequent stage of ranking services that have qualified, the cost parameter (q_1) is considered twice as important than service availability (q_3), which is in turn considered twice as important than response time (q_2). Note that q_4 (security) and q_5 (throughput) are not considered at this stage (the corresponding elements of W are zero), but only services that have met the constraint placed on q_4 have been considered. Formally, the overall “suitability” of a service having a QoS vector equal to (sq_1, sq_2, …, sq_5) is computed as $\sum_{i=1}^{5} sq_i * w_i$, where w_i is the corresponding element of quality vector W.

Vectors *MAX*, *MIN* and *W* are provided as an integral part of the invocation request to the ASOB framework (packed into the HTTP or SOAP headers), and they are considered when a replacement service needs to be selected. Thus, the BPEL designer or the BPEL execution platform arranges for appropriately filling these vectors and setting the request headers accordingly. We will revisit the issue of how quality vectors can be accommodated in the request headers in section 5 and in section 6.2.

5. 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 2, 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 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.

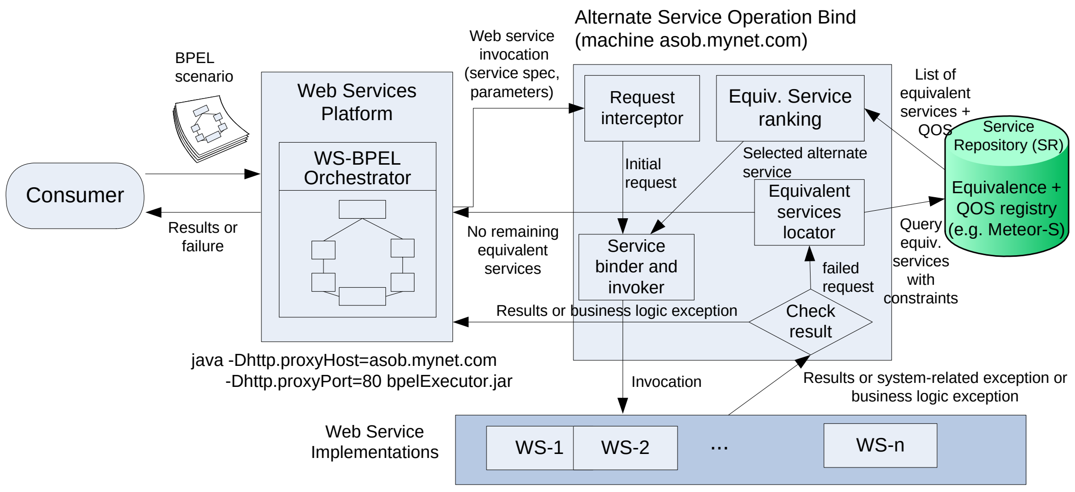

Alternate Service Operation Bind
(machine asob.mynet.com)
BPEL scenario
Web service invocation (service spec, parameters)
Web Services Platform
WS-BPEL Orchestrator
Consumer
Results or failure
Request interceptor
Equiv. Service ranking
List of equivalent services + QOS
Service Repository (SR)
Equivalence + QOS registry (e.g. Meteor-S)
Initial request
Selected alternate service
Equivalent services locator
No remaining equivalent services
Service binder and invoker
Query equiv. services with constraints
failed request
Results or business logic exception
Check result
java -Dhttp.proxyHost=asob.mynet.com -Dhttp.proxyPort=80 bpelExecutor.jar
Invocation
Results or system-related exception or business logic exception
Web Service Implementations
WS-1
WS-2
...
WS-n

Figure 2. ASOB architecture in a proxy-based setup

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 2, 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 or settings in the BPEL component (JBI).

```
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", "Constraints= MIN(0, 0, 3, 4, 0), MAX(2, 0, 0, 5, 0), W=(0.5, 0.1, 0.2, 0, 0)");
```

Listing 5. Applying proxy settings programmatically

The same technique, i.e. setting system properties to modify certain aspects of the HTTP requests behavior, can be used to include in HTTP requests information regarding the constraints and criteria that should be employed by ASOB when selecting the replacement services. A suitable HTTP header to host this information is *user-agent*, which can be set through the *http.agent* system property in Java-based environments; this header typically contains information about the user agent originating the request, and is otherwise unused in web services environments. Setting the *user-agent* header by programmatically assigning a value to the *http.agent* system property is illustrated in Listing 5.

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 3. 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 [43].

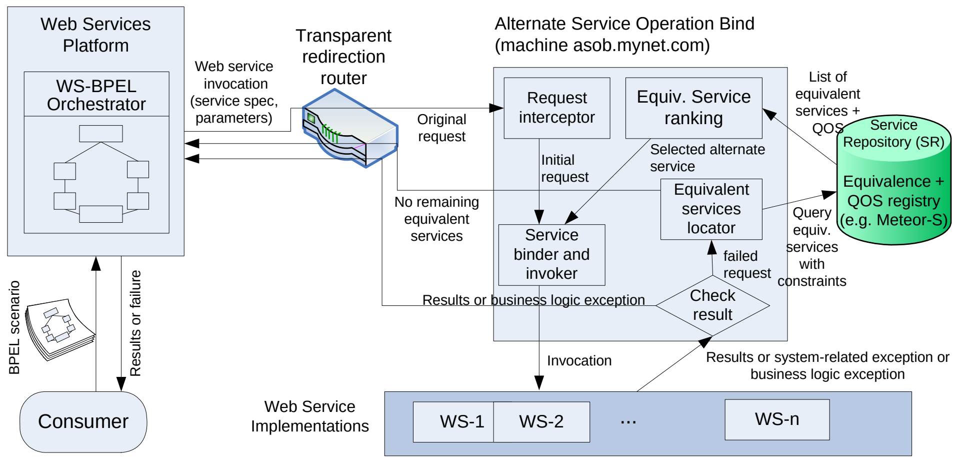

Web Services Platform
WS-BPEL Orchestrator
Web service invocation (service spec, parameters)
Transparent redirection router
Original request
Alternate Service Operation Bind (machine asob.mynet.com)
Request interceptor
Equiv. Service ranking
List of equivalent services + QOS
Service Repository (SR)
Equivalence + QOS registry (e.g. Meteor-S)
Initial request
Selected alternate service
Equivalent services locator
Query equiv. services with constraints
No remaining equivalent services
Service binder and invoker
failed request
Check result
Results or business logic exception
BPEL scenario
Results or failure
Invocation
Results or system-related exception or business logic exception
Consumer
Web Service Implementations
WS-1
WS-2
...
WS-n

Figure 3. ASOB architecture in a redirection router setup

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.

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

5.1 ASOB Components and Functionality

The ASOB module consists of five components, discussed in the 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. RI also extracts the headers designating the service policy, i.e. the vectors MIN, MAX, and W and saves them for later perusal, in case that an exception is raised during the service invocation.
- 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 SBI component includes the XSLT Transformer sub-module, needed for addressing interface mismatches between the original web service specification and the alternate equivalent web service's one, which may occur if the original service fails and the chosen replacement service is only semantically (but not syntactically) equivalent to the originally specified one.
- the Equivalent Service Locator module (ESL). ESL is the component which discovers web services that are functionally equivalent 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, and produces the replacement candidate services list (RCSL). ESL also performs *service filtering*, to guarantee that only services meeting the policy constraints (i.e. the specifications provided in the MIN and MAX quality vectors) appear in RCSL.
- 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 semantically equivalent groups of services using semantic tagging (as, for example METEOR-S and WSMX) or any other suitable approach and should, within each group of semantically equivalent services, identify the sub-groups of *syntactically equivalent* services. If the repository, additionally to equivalence, provides information on the Quality of Service (QoS) characteristics of the services, and ASOB exploits this information to resolve exceptions according to the specified policy (vectors MIN, MAX and W). The QoS characteristics of the services, and particularly those reflecting the services' observed behavior (e.g. response time, availability, successful execution rate etc), can be updated by the SBI module, according to the results of the invocations performed by SBI. More details on the structure of SR are given in section 5.3.
- the Equivalent Service Ranking component (ESR). ESR module is responsible for sorting the equivalent service list according to QoS criteria, in order to adapt the exception resolution process to the specifications given through the policy. The ranking algorithm employed by ESR uses the formula given in section 4.2.2 to compute the overall "suitability score" for each candidate service and then sorts the resulting list in descending order of this score.

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.

5.2 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 2], or the network packets comprising the request will be sent to the transparent redirection router, which will forward them to the ASOB module [Figure 3]. 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. RI also extracts the headers designating the service policy, i.e. the vectors MIN, MAX, and W and saves them for later perusal, in case that an exception is raised during the service invocation. 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 (step 5), 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 (step 5) 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 (e.g. a prolonged network partitioning may have rendered unavailable the name servers responsible for resolving the host name of some services' endpoints, thus these services will be blacklisted, but they will need to be reinstated when the partitioning will be resolved).
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 or the *Fault Code* is one of *Client* (SOAP v1.1) or *Sender* (SOAP v1.2), the reply constitutes an application-level exception which must be passed back to the WS-BPEL script, where it may be 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.
   d) If the reply is neither a valid *output message* nor a declared fault, nor the *Fault Code* is one of *Client* or *Sender*, 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 case additionally includes the fault codes *VersionMismatch*, *MustUnderstand*, *Server* (SOAP v1.1), *DataEncodingUnknown*, *VersionMismatch*, *MustUnderstand*, *Receiver* (SOAP v1.2). These fault codes indicate that the web service invocation has failed due to a systemic error (e.g. the request could have succeeded if it had been sent to a server with updated libraries [44]) and may thus be resolved by invoking an equivalent service, as is the case with other transient errors. Details about SOAP v1.1 and SOAP v1.2 fault codes can be found in the protocol specifications [45], [46].
5. This stage of processing is reached if the original web service invocation has failed due to a system error, and includes the activities taken to resolve it. ESL locates equivalent web service operations, by issuing a query to SR. The methodology to identify syntactically

or semantically equivalent services is outside the scope of this paper, and any pertinent technology (e.g. [12], [13] and [14]) can be used. For the purposes of our work, it suffices to have the ability to present the repository with a service's endpoint and QoS-related criteria, and receive a response containing a list of endpoints of services that (a) are equivalent to the one presented to the repository and (b) satisfy the QoS-related criteria. Each service endpoint in the response will be complemented with (i) its QoS characteristics (ii) the XSLT stylesheets that can be used to transform the payload within the original request to the payload format expected by the replacement service, and the replacement service's reply to the format produced by the originally specified service (and expected by the BPEL scenario).

Having received the alternative services list, the ESR component computes the "suitability score" of its elements using the formula of section 4.2.2 and sorts the list in descending order of this score. If the request does not include a policy specification, a default one, specified by the administrator, is employed. This policy may include only a single QoS attribute (e.g. response time, in which case services with smaller response time will be placed first) or multiple QoS attributes (e.g. [(70% * response_time) + 30% * availability), emulating thus a W vector equal to (0, 0.7, 0.3, 0, 0)].

Finally, ESR iterates over the alternate services list, starting from the first (the "most preferred" one has been placed there) and moving towards the last ("least preferred"). For each such service $S_{candidate}$, the XSLT Transformer module of SBI component adjusts the original request payload to the one required by $S_{candidate}$'s specification; 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 computing the equivalent services list anew. Note that XSLT transformations will also be applied to the reply message as well, to align its format to the one produced by the originally specified service. 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), 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.

Listing 6 and Listing 7 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.

```
OWS ← getPart(request, "webServiceEndPoint");
payload ← getPart(request, "payload");
ReqPolicy ← getPart(request, "policyHeaders")
owsRS ← invoke_bind_WS(OWS, payload);
if (not timeout_occured)
    if (is_normal_reply(owsRS))
        return owsRS to BPEL execution environment; /* request has concluded */
    else /* an exception has been raised */
        /* Initialize list of known exceptions */
        knownEXC ← getExceptionsFromWSDL(OWS);
        if (owsRS.faultCode = "Sender" or owsRS.faultCode = "Client" or owsRS in knownEXC)
            /* Known, application-logic exception, return it */
            return owsRS to BPEL execution environment; /* request has concluded */
`       end if
    end if
end if
/* this point is reached if an exception that can be resolved through alternate service invocation has occurred: timeout, no route to host, fault code type of 'VersionMismatch', etc */
/* returns equivalent services to OWS sorted and filtered, depending on what policy criterion has been applied*/
ewsLIST ← getEquivServices (OWS, ReqPolicy.MIN, ReqPolicy.MAX);
if (ReqPolicy == null)
    weightVector = ASOB.defaultWeights
else
    weightVector = ReqPolicy.W
endif
sortedEWSlist ←sortEquivServices(ewsLIST, weightVector);
ewsRS ← invoke_alternate_ws(OWS, sortedEWSlist, payload);
finRS ← ewsRS;
return finRS to BPEL execution environment;
```

Listing 6. Main request processing in ASOB

```
function invoke_alternate_ws (OWS, ewsLIST, payload)
    while ewsLIST not empty
        EWS ← first_element(ewsLIST)
        payloadXSLT ← fetch XSLT for transforming OWS payload to EWS payload
        transformedPayload ← XSLTransform (payloadXSLT, payload);
        ewsRS ← invoke_bind_WS(EWS, transformedPayload)
        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 */
                replyXSLT ← fetch XSLT for transforming EWS reply to OWS reply
                ewsRS ← XSLTransform ( replyXSLT, ewsRS)
                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(EWS, payload)
    performBinding(EWS);
    result ← invoke(EWS, payload);
    if (timeout_occured)
        update_SR_QoS(WS, NULL, Unavailable, Transient);
    else if (is_normal_reply(result) or result in knownEXC)
        update_SR_QoS(WS, responseTime, Available, NULL);
    else if ((result == UnknownSrv) or (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
    return result;
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 7. Invocation routines

5.3 Service Repository

As discussed in the previous sections, the service repository should host all information required to perform certain operations in ASOB and more specifically:

1. to identify services equivalent to the one originally invoked
2. to determine the QoS aspects of each service
3. to get the list of known exceptions for each service.
4. to retrieve the XSLT stylesheets that can be used to transform request payloads and reply messages for services that are semantically but not syntactically equivalent.

In the current implementation of ASOB, this information is hosted in a MySQL database, whose schema is depicted in Figure 4. In this schema, services are classified into *semantically equivalent* groups, which are further subdivided into *syntactically equivalent* groups. Classification in semantically equivalent groups enables the identification of services that offer the same functionality, and thus can be used as replacements for one another. Classification to syntactically equivalent services is used for optimization purposes in a threefold manner:

1. Consider the case where some operator O_1 has deployed the syntactically equivalent services *G1 = {S1, S2, S3}*, for reliability or load-balancing purposes. If the original invocation received by ASOB is directed to S1 and this fails, *S2* or *S3* could be selected as replacements. In such a case, there is no need to apply an XSLT transformation to the original payload before invoking the replacement service or to the reply, after it has been received by the replacement service. The classification to *syntactically equivalent* groups enables the detection of such cases and can save the (considerable) cost to perform the XSLT transformations.
2. Consider that, additionally to the above services, some operator O_2 has deployed services *G2 = {S3, S4}* which are (a) semantically equivalent but syntactically different than the services of G1 and (b) syntactically equivalent between them. Without the classification into syntactically equivalent groups, we would need to store XSLT transformations between all service pairs (g_1, g_2), where $g_1 \in$ G1 and $g_2 \in$ G2 [and also for all pairs (g_2, g_1)]. The classification into syntactically equivalent groups enables us to store a single XSLT transformation between G1 and G2 (including both payload and reply transformations) and an additional XSLT transformation for the inverse direction (G2 to G1).
3. Consider, finally, that some original invocation to *S1* has failed, and *S3* is deemed as being the most suitable replacement. Since the services are not syntactically equivalent, an XSLT transformation is applied and *S3* is invoked, but this invocation fails too, and *S4* is the next service to be tried. Since *S4* is syntactically equivalent to *S3*, the payload computed for the previous invocation can be directly used, without the need to reapply a transformation to the original payload. ASOB includes this optimization, by saving pairs of *(syntacticGroupId, computedPayload)* as they are generated along the processing of some request, and reusing an already computed payload when some service of the same *syntacticGroupId* needs to be invoked.

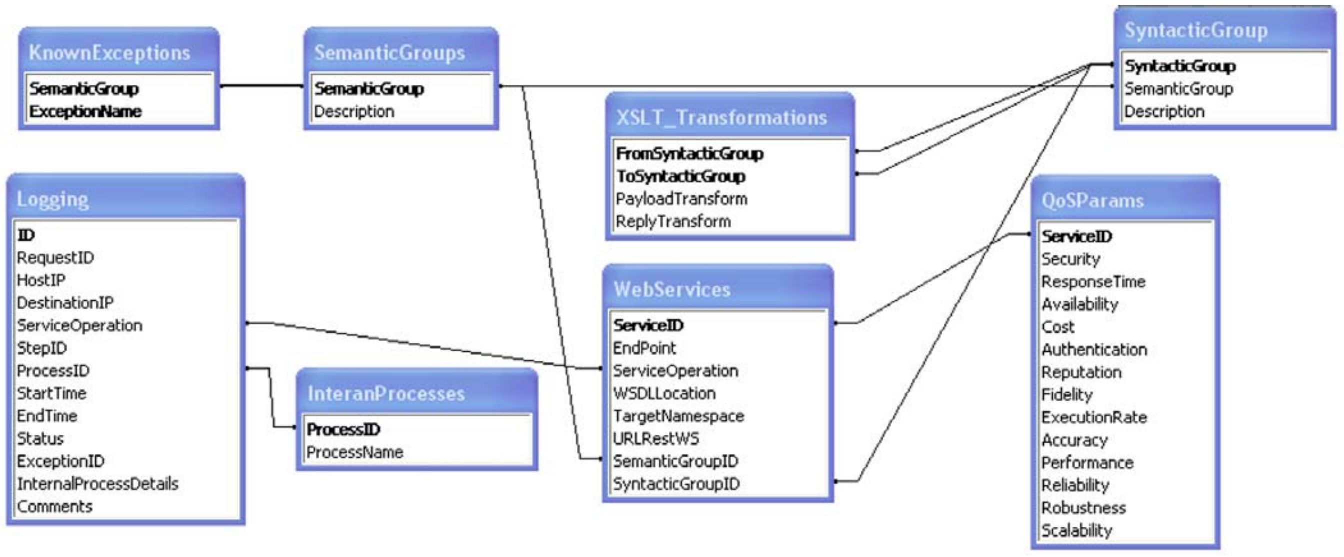


Figure 4. Service repository schema

The QoS parameters of services are in fact stored in the same table as the web services, but shown as a distinct table in Figure 4 for readability purposes. Storage in the same table saves a join operation at the database level. Finally, the *InteranProcesses* and *Logging* tables are used by ASOB to store logging information regarding the status and performance of steps taken to service the intercepted requests. This information is collected for both web service invocations performed by ASOB (and these data are subsequently used to update the services' QoS characteristics), and the internal ASOB operations (which are used for code profiling and optimization).

Considering the sizes of web services' QoS databases that have been made available insofar (e.g. [47]), it is expected that the whole repository will be able to fit into the hosting machine's main memory (except for the tables storing logging information), thus disk I/O can be avoided for the benefit of the performance. In the MySQL environment, this can be accomplished by using the memory (heap) storage engine ([48]).

6. DESIGN AND IMPLEMENTATION CONSIDERATIONS FOR ASOB

In this section we revisit two issues that have been discussed in the previous sections, to discuss further design and implementation options that can be adopted. These issues concern the number of XSTL stylesheets that need to be accommodated in the repository, in order to provide with transformations between semantically but not syntactically equivalent service groups, and the repercussions that this number may have on the manageability of the XSLT repository. The second issue concerns how policy specifications can be accommodated within the request headers of each invocation (either in HTTP- or SOAP-level), to allow for fine-grained policy specification in the context of BPEL processes.

6.1 XSLT Transformations: Manageability vs. Efficiency

Recall from section 5.3 that services within the repository are classified into categories of *semantically equivalent* services *SE1, SE2, …, SEn*, and each such category SEi is further subdivided into groups of syntactically equivalent services *SEi(G1), SEi(G2), …, SEi(G*$_{\#SEi}$*)*, where *#SEi* is the number of syntactically equivalent service groups within the category of semantically equivalent services SEi. In order for ASOB to be able to perform payload and reply transformation between all semantically equivalent services (so that any service within a category SEi can be substituted by another service in the same category, despite any syntactic differences that may exist between them), the repository SR must store XSLT stylesheets to cater for transformation of payloads and replies between all pairs of groups *SEi(G1), SEi(G2), …, SEi(G*$_{\#SEi}$*)*. Therefore, the number of XSLT stylesheets that must be accommodated in the repository for a specific category SEi is

$\#xslt(SEi) = 2 * (\#SEi^2 - \#SEi)$

(the number of permutations of the groups of *SEi* taken two at a time, multiplied by two to cater for both payload and reply XSLTs), and the overall number of XSLT stylesheets in the repository is

$$\#xslts = \sum_{i=1}^{n} \#xslt(SEi) = \sum_{i=i}^{n} 2 * (\#SEi^2 - \#SEi) = 2 * \sum_{i=i}^{n} (\#SEi^2 - \#SEi)$$

where *n* is the number of categories of semantically equivalent services. From the formulas above, it follows that the number of XSLT stylesheets increases linearly with the number of semantically equivalent categories, but exhibits an increase as a square of the number of syntactically equivalent groups within categories of semantically equivalent services. Besides the increased space requirements that a large repository will have in this case (which may preclude the storage of the whole repository into main memory), this arrangement leads to considerably high repository administration costs: when a new group of syntactically equivalent services SEi(G_{new}) is introduced into a category SEi already numbering *#SEi* groups, the repository administrator must craft *4*#SEi* new XSLT stylesheets, to cater for the payload/reply transformations between the newly defined group and each of the existing ones. In a context where *#SEi* may be high, this requirement will undermine the potential to add new service groups to the repository.

To decrease the number of XSLT schemes in database, a *message canonical form* approach may be employed. According to this approach, for each category of semantically equivalent services the repository administration defines a *payload canonical form* and a *reply canonical form*, which effectively includes placeholders for any parameter that services of this category accept in the payload or return in the reply, respectively. Under this arrangement, to transform the payload of group G1 to that of G2, an XSLT transformation *G1_Payload_to_Canonical* should be applied, followed by the transformation *G2_Payload_from_Canonical*. The number XSLT stylesheets that must be accommodated in the repository for a specific category SEi according to this approach is

$\#xslt_{canon}(SEi) = 4 * \#SEi$

and the overall number of XSLT stylesheets in the repository is

$$\#xslts_{canon} = \sum_{i=1}^{n} \#xslt_{canon}(SEi) = 4 * \sum_{i=i}^{n} \#SEi$$

The number of the XSLT stylesheets required for each category here increases linearly with the number of groups within the category, and is always less than the number of XSLT stylesheets required in the direct transform approach, except for the case that a category includes exactly two groups, where the direct transform approach requires four stylesheets and the canonical form approach requires eight; also the number of stylesheets needed for categories including exactly three groups is the same for both approaches and equal to 12.

In terms of repository administration overhead, when a new group of syntactically equivalent services is introduced in a category, it suffices to define XSLT stylesheets to transform the payload and the reply expected and produced by this group to and from the category canonical forms, i.e. four XSLT stylesheets in total. Figure 5 presents an example of payload canonical form for a *PersonLocator* service (b), and two forms of payloads bearing the same parameters, formatted according to the requirements of specific groups. The XSLT stylesheet mapping between a specific payload format and the canonical form need to perform tag substitution as appropriate and insert/remove intervening syntax elements (as is the case of the *FullName* tag).

```
<PersonalInfo>
  <Name>John</Name>
  <Surname>Smith</Surname>
  <Country>USA</Country>
</PersonalInfo>
```
(a)

```
<TYPE>
  <param1>John</param1>
  <param2>Smith</param2>
  <param3>USA</param3>
</TYPE>
```
(b)

```
<PersonDetails>
  <FullName>
    <FirstName>John</FirstName>
    <LastName>Smith</LastName>
  </FullName>
  <Country>USA</Country>
</PersonDetails>
```
(c)

Figure 5. Two forms of *PersonLocator* service payloads (a and c) and the payload canonical form for this category (b)

The canonical form approach reduces effectively the overall number of stylesheets that need to be defined and hosted in the repository, but introduces the need for an additional XSLT transformation, when a message (payload or reply) needs to be reformatted. This overhead is quantified in the performance evaluation in section 7.

When ASOB needs to transform some payload to another format in order to invoke an alternate service, it saves the canonical form of the payload that has been produced in the intermediate step, thus if the alternate service invocation fails too, subsequent transformations can use the already computed canonical form as a starting point. In this respect, the overall number of transformations that must be applied increases only by at most two, as compared to the approach of having direct transformations between all groups of syntactically equivalent services, instead of being doubled (one extra transformation to transform the initial payload to the canonical form, and one extra transformation to transform the final reply to the canonical form; the latter can be saved, if the service that produces the final reply happens to be in the same group of syntactically equivalent services as the originally invoked one, or if a business logic exception has been raised).

Table recapitulates the differences between the direct XSLT transformation approach and the canonical form approach, in terms of overall number of XSLT stylesheets (with impact on storage space), number of XSLT transforms and administration overhead.

	Direct XSLT transformation approach	Canonical form approach
Number of XSLT stylesheets per category of semantically equivalent services	$2 * (\#SEi^2 - \#SEi)$	$4 * \#SEi$
Overall number of XSLT stylesheets	$2 * \sum_{i=i}^{n} (\#SEi^2 - \#SEi)$	$4 * \sum_{i=i}^{n} \#SEi$
Number of payload transformations	Equal to the number of the groups containing the alternate services that are tried	Equal to the number of the groups containing the alternate services that are tried plus one
Number of reply transformations	0, if the result is returned by a service syntactically equivalent to the originally specified one, or an exception is raised; otherwise 1	0, if the result is returned by a service syntactically equivalent to the originally specified one, or an exception is raised; otherwise 2
Number of XSLT stylesheets need to be defined when introducing a new group of syntactically equivalent services	$4 * \#SEi$	4

Table 2. Comparison of direct XSLT transformation approach and canonical form approach

6.2 Accommodating Policy Specifications in Request Headers

Although the ASOB middleware provides means for extracting QoS policy specifications from the headers of the web service invocation requests it receives, the placement of the appropriate policy specifications in these headers is not always a trivial task. When web service invocations are written in a general-purpose language, such as Java or C#, the programmer may use constructs of the programming language to define the appropriate values for the headers, e.g. in Java it suffices to use the *header.addHeaderElement* method. The current BPEL standard, however, does not include similar provisions, thus the BPEL scenario designer may have less control over the specification of the QoS policy for exception handling.

Under the java-based proxy architecture (Figure 1), it is possible to specify the desired QoS policy through the *http.agent* property of the Java environment, which effectively sets the *user-agent* HTTP header; though this approach is viable, it presents the limitation that all BPEL process instances running in deployment environment will be executed under the same policy preferences, determined by BPEL designer at the deployment/BPEL container configuration time. This technique is independent from any particular BPEL process engine execution environment. BPEL container configuration can be also applied in the transparent router architecture of Figure 2, specifying that a particular header with a pre-specified value will be added to all outgoing invocations. Again, a policy defined in this manner applies to all BPEL process instances, with no ability for a more fine-grained specification (e.g. different policies for different executions, or different policies for distinct service invocations). The ASOB middleware also includes a default policy, for requests that are not supplied with one.

However, many vendors have recognized the need to allow a BPEL process designer to control the headers for each particular outgoing request and have added extra functionality in BPEL design environments, allowing the designer to set and examine request and response headers, either at SOAP or HTTP level. This ability can be exploited by BPEL designers to specify a particular policy to be in effect throughout the execution of the scenario (typically by adding the appropriate code at the beginning of the scenario) or even different policy preferences per web service invocation within a BPEL process instance execution (by preceding the invocation with suitable code). Listing 8 presents how the OpenESB sxnmp extension [7] can be used for copying the policy specified in the headers of the BPEL scenario execution request to the headers of a particular web service invocation made in the context of the scenario, while Listing 9 illustrates the same provisions of the Oracle BPEL Manager bpelx extension [5]. This functionality has been also integrated in GUI-based BPEL scenario designers, such as NetBeans NM, depicted in Figure 6.

```
<copy>
  <from variable="policyVectorsInSOAPHeaders" sxnmp:nmProperty="org.glassfish.openesb.headers.soap"/>
  <to variable="var1" sxnmp:nmProperty="org.glassfish.openesb.headers.soap"/>
</copy>
<copy>
  <from variable="policyVectorsInHTTPHeaders" sxnmp:nmProperty="org.glassfish.openesb.inbound.http.headers"/>
  <to variable="var2" sxnmp:nmProperty="org.glassfish.openesb.outbound.http.headers"/>
</copy>
```
Listing 8. OpenESB sxnmp extension for setting outgoing request headers

The next figure illustrates graphical representation of the above listing.

```
<invoke name="Invoke_1" partnerLink="partnerLink"
    portType="portType" operation="process"
    inputVariable="Invoke_1_process_InputVariable"
    outputVariable="Invoke_1_process_OutputVariable"
    bpelx:inputHeaderVariable="policyVectorInSOAPHeader" />
```
Listing 9. Oracle BPEL Manager bpelx extension for setting outgoing request headers

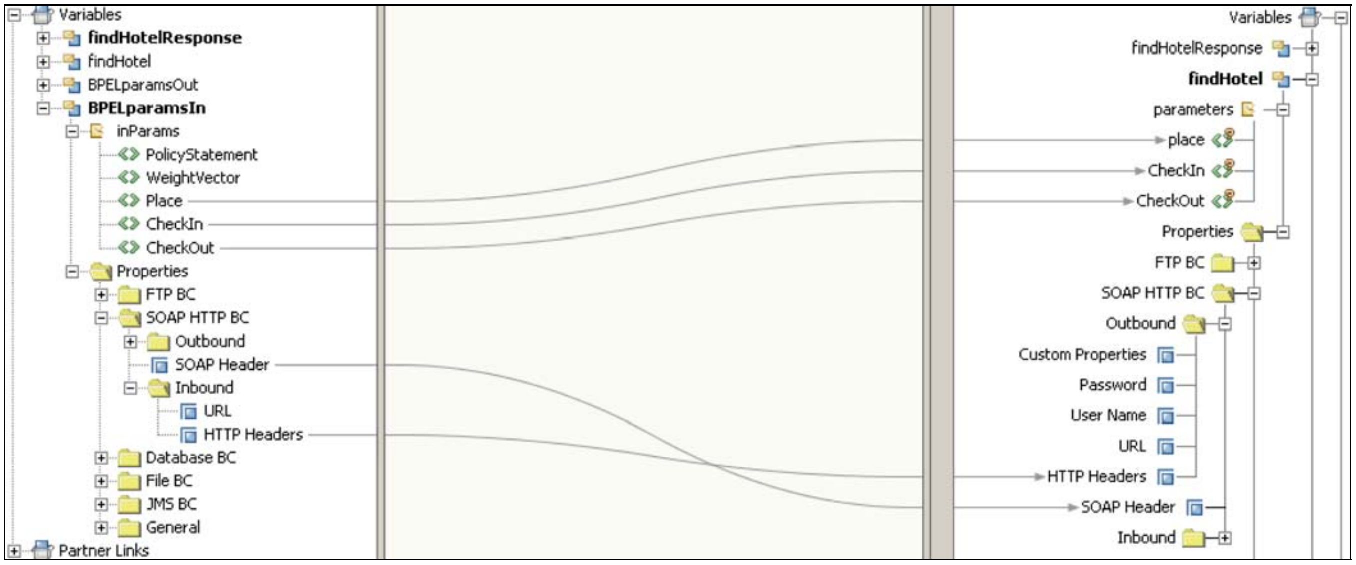


Figure 6. Netbeans NM Properties

7. 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 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.

7.1 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 [49] and executed using the WS-BPEL 2.0 compliant Java Business Integration (JBI) component [50], deployed on Glassfish v2 application server [7]. 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* [51] 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 ASOB to retrieve and sort the list of equivalent services, and to apply XSLT transformations. 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 and application of XSLT transformations - 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.

7.2 Performance Analysis

Figure 3 illustrates the ASOB internal process time against the database size (denoted as *db-size*) and the number of (semantically) 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, the time needed for the extra network message transmission (from the BPEL script to the middleware and the XSLT Transformation durations (the time cost of XSLT transformations will be discussed later in this section). 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

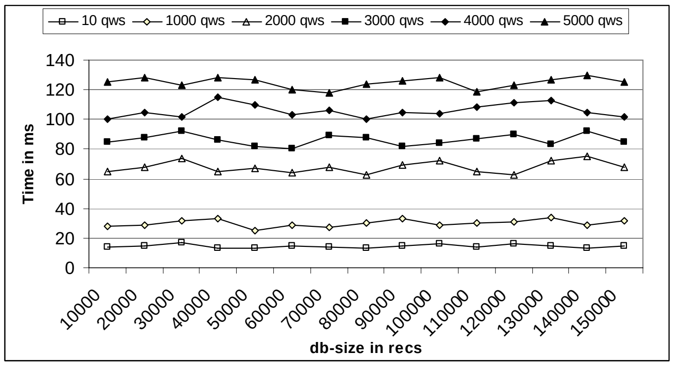


Figure 7. ASOB internal process time

Figure 7 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.

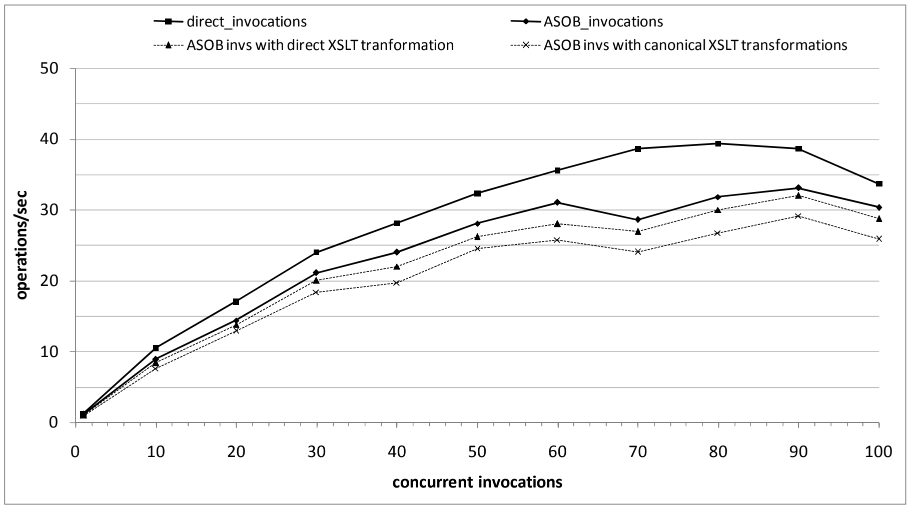


Figure 8. ASOB-mediated vs. direct WS invocation throughput

Figure 8 illustrates the number of invocations that can be served in a unit of time against the number of concurrent invocations when (a) web services are directly invoked, (b) when invocations are made through the ASOB middleware (c) using ASOB with one direct XSLT transformation and (d) using ASOB with one XSLT transformation, according to the canonical form approach. The metrics depicted in this diagram have been obtained by directly invoking the web services through a Java client, and not through a BPEL scenario (BPEL scenarios are considered in the following diagrams). Case (a) exhibits a throughput drop of 8-19%, except for the case of 70 concurrent invocations, where the drop is approximately 25%. Accordingly, in case (c) the drop of is approximately 11-22% (except for the point of 70 concurrent invocations where a drop of 30% is observed), while for case (d) the drop is in the range 13%-29% (again, for the point of 70 concurrent invocations, a drop of 36% is exhibited). The performance drops in the ASOB-mediated cases are 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 (without the 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.

Furthermore, 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 direct and the ASOB-mediated approaches are small to medium [8%-12% in case (b), 10%-16% in case (c) and 13%-19% for case (d)].
2. from 50 to 80 aINV, where the performance differences are higher [12%-25% in case (b), 16%-30% in case (c) and 19%-36% for case (d)].
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.

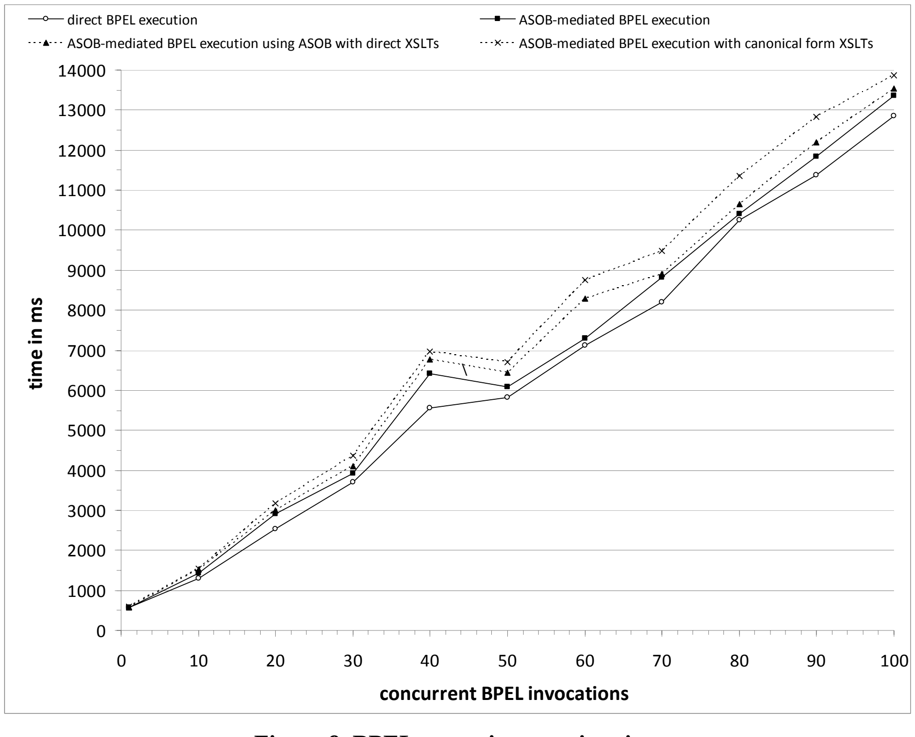


Figure 9. BPEL scenario execution time

Figure 9 illustrates the overall execution 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 are 100msec, 100msec and 300msec, respectively. The x-axis represents the concurrent BPEL process executions (invocations, through the BPEL web service interface) and y-axis the time passed until all of them completed successfully. Similarly to the previous diagrams, we consider the following cases: (a) direct web service execution [without the intermediation of ASOB] (b) ASOB-mediated web service execution without XSLT transformation (c) ASOB-mediated web service execution with direct XSLT transformations and (d) ASOB-mediated web service execution with canonical form XSLT transformations. It is obvious that the BPEL process time is slightly increased in the ASOB-mediated cases, but the increment is small – less than 6% without XSLT, less than 8% with direct XSLT and less than 10% with canonical form XSLT, except for two anomaly points (20 and 40 concurrent invocations) where the observed overheads are 14 and 15% for case (b), 18% and 21% for case (c) and 25% and 28% for case (d), owing probably to reasons as those listed for Figure 8 above. Additionally, in case of 60 concurrent invocations the ASOB-mediated cases involving XSLT transformations exhibit increased times over the non ASOB-mediated case (16% and 21% for direct and canonical form transformations, respectively), for the same reasons explained in the previous figure.

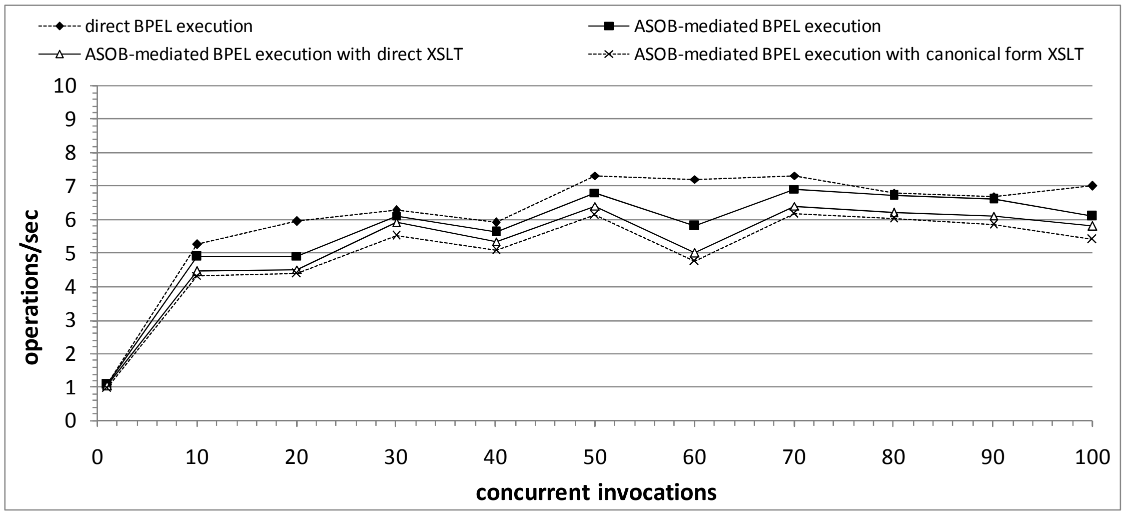


Figure 10. ASOB-mediated vs. direct invocation BPEL scenario execution throughput

Figure 10 depicts the BPEL scenario execution throughput against the number of concurrent invocations for cases (a), (b), (c) and (d) described above. The behavior is consistent with the previous diagrams, i.e. the throughput drop is in the range of 6%-10% in case (b), 7%-12% in case (c) and 9%-20% in case (d). Again, two anomaly points (20 and 60 concurrent invocations) are observed, for which performance drops are quantified to 18% and 20% in case (b), 25% and 29% in case (c) and 25 and 31% in case (d). Again, reasons as those listed for Figure 8 above may explain these performance 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, consistently to the case of individual web service invocations (Figure 8) .

Figure 11 focuses on the performance aspects of the two XSLT transformation approaches, i.e. the direct XSLT transformation and the canonical XSLT transformation approaches. The benchmark performed for this case involves the execution of a BPEL scenario, in the context of which an unavailable service is invoked. The exception is caught by the ASOB middleware and is resolved by invoking an alternate service belonging to a different syntactic group, necessitating thus one syntax alignment operation, which maps to two XSLT transforms in the direct approach (one for the payload and one for the reply) and four XSLT transforms for the canonical form approach (two for the payload and two for the reply). The experiment was repeated, setting ASOB to select an unavailable service as a first replacement, thus the exception resolution process in ASOB would move on to the next replacement service (which succeeded), resulting overall in three invocation attempts. All invoked services were selected to belong to different syntactic groups, thus two syntax alignment operations were needed for the payload and one for the reply. Finally, the experiment was repeated once more, arranging for the first two replacement services to fail and the third one to succeed (an overall of four invocations), necessitating thus three alignment operations for the payload and one for the reply (again, all invoked services were selected to belong to different syntactic groups). The number of concurrent invocations, as in the previous experiments, ranged from 10 to 100.

As we can observe in the diagrams, the canonical form approach exhibits a higher overhead than the direct approach, and this is expected, since more XSLT transformations are involved for a single syntax alignment operation. The performance overhead appears to be equal to the cost of performing two XSLT transforms in the first area of the diagram (10-40 concurrent operations; the average cost of performing an XSLT transform in the testbed used was approximately 17msec), but beyond that area, and especially beyond the point of 70 concurrent executions, the overhead appears to rise more sharply. This can be explained by considering that XSLT transforms are essentially CPU-bound operations, and beyond the point of 40 concurrent operations, the contention of the different threads executing XSLT transforms for the CPU becomes apparent. Below this point the contention is not observed, and this can be explained if we consider that the process of serving a web service execution request involves bursts of CPU-bound operations (mainly sorting the candidate service list and performing XSLT transforms) and network-bound operations (expecting requests for service invocation or invocation results). Therefore, while some execution threads are blocked on network operations, the others can perform their CPU-related tasks, and up to the point of 40 concurrent operations the CPU has amble power to service all threads that are not blocked; beyond that point, the CPU is saturated and threads that are ready to perform CPU-bound tasks are forced to wait for the CPU to be allocated to them.

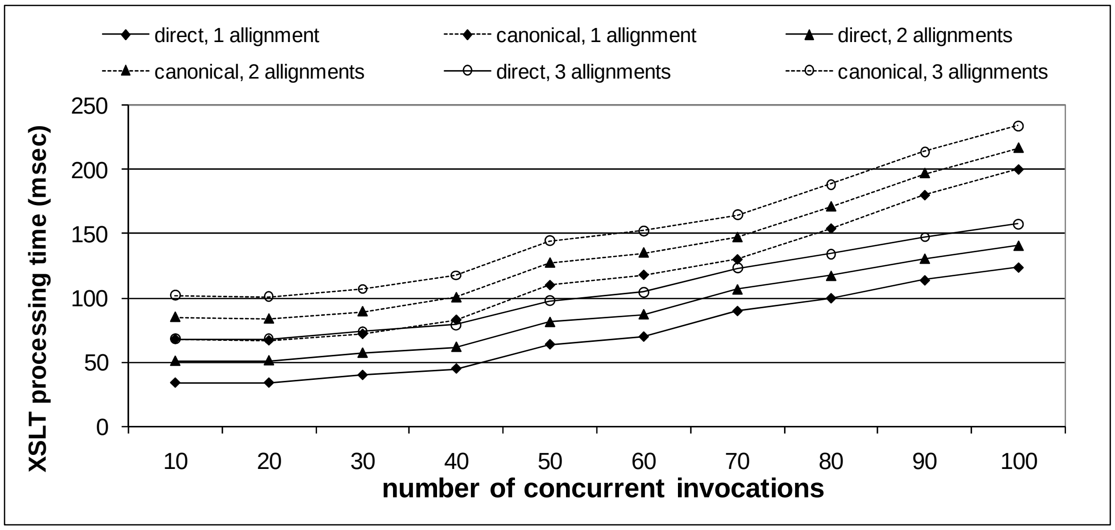


Figure 11. XSLT processing time for direct and canonical form approach

8. 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. Additionally, the middleware caters for bridging syntactic differences between functionally equivalent services, allowing thus a wider selection of replacement services. Syntactic differences are handled by applying XSLT transformations, and the ASOB platform maintainers can opt here for one of two proposed approaches, considering the performance vs. manageability tradeoffs of these approaches. Finally, the ASOB middleware can exploit extensions already available in certain BPEL environments, according to which designers may set and examine request headers, both at HTTP and SOAP level; this functionality can afford the specification of service replacement policies in a more fine-grained fashion, either per BPEL scenario execution or per individual service invocation.

One 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. A further area of research is the optimization of the XSLT transforms in the canonical form approach, by allowing the repository administrator to specify the transformations through the canonical form, and having the platform pre-compute a XSLT transformations that would directly transform payloads and replies from one syntactic group to another, without creating the intermediate canonical form.

9. 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] Orchestra, Open Source BPEL Solution, http://orchestra.objectweb.org/xwiki/bin/view/Main/WebHome

[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] L. Zeng, B. Benatallah, A.H. Ngu, M. Dumas, J. Kalagnanam, H. Chang. QoS-aware middleware for web services composition. IEEE Trans. Softw. Eng., 30(5), 2004.

[13] L. Zeng, Dynamic Web Services Composition, PhD thesis, Univ. of New South Wales, 2003.

[14] H.C.-L and, K. Yoon, Multiple Criteria Decision Making, Lecture Notes in Economics and Mathematical Systems. Springer-Verlag, 1981.

[15] O. Moser, F. Rosenberg, S. Dustdar, Non-Intrusive Monitoring and Service Adaptation for WS-BPEL, WWW 2008, Beijing, China, pp. 815-824.

[16] Active Endpoints. ActiveBPEL Engine, 2007. http://www.active-endpoints.com/.

[17] F. Baligand, N. Rivierre, T. Ledoux. A Declarative Approach for QoS-Aware Web Service Compositions. B. Kramer, K.-J. Lin, and P. Narasimhan (Eds.): ICSOC 2007, LNCS 4749, pp. 422–428, 2007.

[18] H. Cao, H. Jin, S. Wu, L. Qi, ServiceFlow: QoS Based Service Composition in CGSP. Proceedings of IEEE EDOC'06.

[19] Liangzhao Zeng; Hui Lei; Jun-jang Jeng; Jen-Yao Chung; Benatallah, B. Policy-driven exception-management for composite Web services, E-Commerce Technology, Proceedings of CEC 05, 19-22 July 2005, pp. 355 – 363

[20] C. Kareliotis, C. Vassilakis, E. Rouvas, P. Georgiadis, Exception Resolution for BPEL Processes: a Middleware-based Framework and Performance Evaluation. Procs of iiWAS 2008, Linz, Austria.

[21] Liangzhao Zeng, Jun-Jan Jeng, Santhosh Kumaran and Jayant Kalagnanam, Reliable Execution Planning and Exception Handling for Business Process, LNCS, Springer, Technologies for E-Services, 2003. p.119-130

[22] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, Maria Luisa Villani, QoS-Aware Replanning of Composite Web Services Proceedings of the IEEE International Conference on Web Services 2005, Pages: 121 – 129

[23] A. E. Arpacı, A. B. Bener. Agent Based Dynamic Execution of BPEL documents. Proceedings of ISCIS 2005, LNCS 3733, pp. 332 – 341, 2005.

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

[25] 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

[26] 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/

[27] 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)

[28] The OWL Services Coalition, OWL-S: Semantic Markup for Web Services, Available at: http://www.daml.org/services/owl-s/1.0/owl-s.html.

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

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

[31] Derong Shen, Ge Yu, Tiezheng Nie, Rui Li, and Xiaochun Yang. Modeling QoS for Semantic Equivalent Web Services. Advances in Web-Age Information Management, LNCS 3129, Springer Berlin / Heidelberg, 2004, pp. 478-488.

[32] Daniel A. Menasc´e. Mapping service-level agreements in distributed applications. IEEE Internet Computing, 8(5):100–102, 2004.

[33] Daniel A. Menasc. Qos issues in web services. IEEE Internet Com-puting, 6(6):72–75, 2002.

[34] B. Sabata, S. Chatterjee, M. Davis, J. Sydir, and T. F. Lawrence, Taxomomy of QoS Specifications, Proceedings of the 3rd Workshop on Object-Oriented Real-Time Dependable Systems (WORDS '97), February 1997.

[35] WSLA Project Web site: http://www.research.ibm.com/wsla.

[36] J. O`Sullivan, D. Edmond, and A. Ter Hofstede: What is a Service?: Towards Accurate Description of Non-Functional Properties, Distributed and ParallelDatabases, 12:117-133, 2002.

[37] S. Rajesh and D. Arulazi: Quality of Service for Web Services-Demystification, Limitations, and Best Practices, March 2003, Available at: http://www.developer.com/services/print.php/2027911

[38] Amit Sheth, Jorge Cardoso, John Miller and Krys Kochut, QoS for Service-oriented Middleware. In Proceedings of the Conference on Systemics, Cybernetics and Informatics, 2002

[39] Le-Hung Vu, Manfred Hauswirth, and Karl Aberer. QoS-Based Service Selection and Ranking with Trust and Reputation Management. R. Meersman and Z. Tari (Eds.): CoopIS/DOA/ODBASE 2005, LNCS 3760, pp. 466–483, 2005

[40] Xia Wang, Tomas Vitvar, Mick Kerrigan, and Ioan Toma. A QoS-aware Selection Model for Semantic Web Services. Service-Oriented Computing – ICSOC 2006, pp. 390-401.

[41] Al-Masri, E., and Mahmoud, Q. H.. Discovering the best web service. Poster presentation in the 16th International Conference on World Wide Web (WWW), 2007, pp. 1257-1258.

[42] Al-Masri, E., and Mahmoud, Q. H. QoS-based Discovery and Ranking of Web Services. IEEE 16th International Conference on Computer Communications and Networks (ICCCN), 2007, pp. 529-534.

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

[44] Microsoft Corporation. Using SOAP faults. In .NET Development reference, http://msdn.microsoft.com/en-us/library/aa480514.aspx

[45] SOAP v1.1 http://www.w3.org/TR/2000/NOTE-SOAP-20000508/#_Toc478383507

[46] SOAP v1.2 http://www.w3.org/TR/2000/NOTE-SOAP-20000508/#_Toc478383507

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

[48] MySQL. The MEMORY (HEAP) Storage Engine. http://dev.mysql.com/doc/refman/5.0/en/memory-storage-engine.html

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

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

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