# Virtualizing Services and Resources with ProBus: The WS-Policy-Aware Service and Resource Bus*

Ralph Mietzner, Tammo van Lessen, Alexander Wiese, Matthias Wieland,
Dimka Karastoyanova, and Frank Leymann
Institute of Architecture of Application Systems, University of Stuttgart
Universitätsstraße 38, 70569 Stuttgart, Germany
firstname.lastname@iaas.uni-stuttgart.de

## Abstract

*A fundamental principle of service oriented architectures is the decoupling of service requesters and service providers to enable late binding of services at deployment time or even dynamic binding of services at runtime. This is important in enterprise settings, where different services that implement business functions in critical business processes are dynamically chosen based on availability or price. The same problem also applies to dynamic Grid environments where resources need to be dynamically chosen based on availability and other non-functional properties. The WS-Policy framework describes how policies for both providers and requesters are specified to allow the selection of services based on these policies. Existing approaches, using WS-Policy, have drawbacks by placing the burden of the service selection partially on the client. In this paper we present an extended enterprise service bus that allows service clients to submit policies to which service providers need to comply with together in one message with the service invocation request. We show how these policies are evaluated in the bus and how policies are defined for not only stateless services, but also stateful resources.*

## 1. Introduction

Today, enterprises are faced with numerous challenges relating to the flexibility of both operational procedures as well as the supporting IT infrastructure and applications. A frequent solution to cope with the request for flexibility on an IT-level is the adoption of service oriented architectures (SOA). Service oriented architectures allow dynamic binding of services in business processes or service orchestrations. This is possible since service requesters are decoupled from concrete service providers. Decoupling service requesters from concrete services allows exchanging services without modifying the service requesters' applications.

In enterprise settings an enterprise service bus (ESB) provides a level of indirection between service providers and service requesters and thus virtualizes concrete service implementations. This is done by matching the *functional properties* of the interface of the service provider and functional requirements of the service requester. All matching services are put on a candidate list. Additionally to that it can be possible that service clients want to select a service from the candidate list based on *non-functional properties* at runtime. For that a services on the candidate list that fulfills the non-functional requirements has to be determined.

The contribution of this paper is to introduce ProBus, a standards-based ESB capable of *policy-based service and resource selection* in the service bus. Our approach is based on the well-established WS-Policy standard [19] and is particularly suited to dynamic environments where services and resources dynamically appear and disappear and resource properties change frequently. The presented approach reduces implementation complexity for clients, as it delegates the service selection completely to the service bus. Our approach extends the WS-Policy-based selection of services to stateful resources, thus unifying both: service and resource virtualization.

The remainder of this paper is structured as follows: In Section 2 we lay the necessary foundations by showing how dynamic service selection based on WS-Policy is done today and by describing how it is done in ProBus. In Section 3 we introduce a scenario to motivate dynamic resource selection in the bus. In Section 4 we then de-

*The work published in this article was partially funded by the SUPER project under the EU 6th Framework Programme Information Society Technologies Objective (contract no. FP6-026850, http://www.ip-super.org/).

978-0-7695-3709-2/09 $25.00 © 2009 IEEE
DOI 10.1109/ICWS.2009.44



IEEE computer society

Authorized licensed use limited to: UNIVERSITAETSBIBL STUTTGART. Downloaded on June 15,2010 at 13:09:23 UTC from IEEE Xplore. Restrictions apply.

scribe how ProBus handles WS-Policy based resource selection. We describe the architecture and implementation of ProBus which is implemented as an extension to the open source Apache ServiceMix ESB (Section 5) and describe how it relates to related concepts used in enterprise and Grid environments (Section 6). Section 7 finishes the paper with a conclusion and outlook.

## 2. Service Selection Using WS-Policy

WS-Policy [19] is a framework and model to describe policies that are represented by requirements, capabilities and general characteristics in a Web service (WS) world. The WS-Policy specification defines the model and syntax to define such policies. A policy is a collection of *alternatives* that are composed out of a set of *assertions*. WS-PolicyAttachment [18] defines how to attach policies to arbitrary entities in a WS based system. In particular, WS-PolicyAttachment specifies how policies can be attached to elements in the WSDL [20], description of a service. Annotating services with policies allows describing non-functional properties for a service formalized in the policy. Service clients can select services based on functional properties (described in WSDL) and non-functional properties (described in WS-Policy) for example from a UDDI registry. To do so, a service client describes the requested non-functional properties (such as security requirements, transaction support or price) in a requester policy. WS-Policy defines an intersection algorithm to compare requester and provider policies. The intersection algorithm first normalizes the policies and then computes their intersection. The result of the intersection is the so-called *effective policy*. An effective policy is a new policy that contains all alternatives of the requester and provider policies that are compatible. Two alternatives of two policies are said to be compatible if the one alternative contains the same assertions as the other alternative. If at least one alternative of a policy is compatible to any alternative in the other policy, these policies are said to be compatible. This means that in order for two policies to be compatible the effective policy that results from their intersection must contain at least one non-empty alternative. The effective policy can be seen as a contract between requester and provider, such as that the invocation needs to be encrypted in a certain way. WS-Policy as a general framework only compares assertions based on QNames. Whether the attributes of the assertion or any possible child elements are the same in the requester and the provider policy is explicitly left open. WS-Policy considers this to be a domain-specific problem that is to be solved by the so-called *domain-specific processing* [19, Section 4.5].

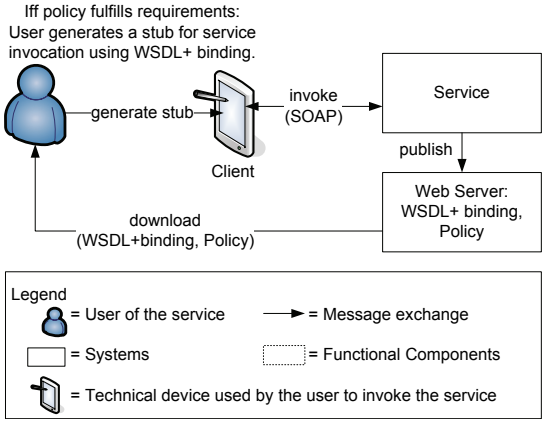


**Figure 1. Class 1: Manual service selection**

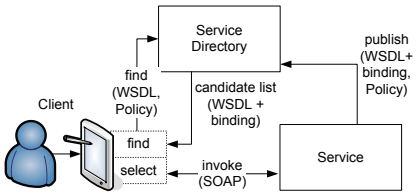


**Figure 2. Class 2: Policy based dynamic service selection**

### 2.1. Different Classes of WS-Policy based Service Selection

In the following the different classes of dynamic service binding are described. For resources these classes are similar only that the dynamic binding is done for every usage of the resource instead of normally once for a service.

**Class 1: Manual service selection** The straight forward way of service selection is to search manually for a service description (WSDL) that provides the functionality and to check if the service properties (Policy) match the needs. This is shown in Fig. 1. Many users prefer that static manual service binding because it has following advantages: (i) it is easy to use, (ii) the user knows exactly which concrete service is used and (iii) no functional components on the client are needed.

**Class 2: Policy based dynamic service selection** However autonomous dynamic service binding is needed for flexibility and scalability reasons. In currently used approaches dynamic binding and policy evaluation is done by the client applications as shown in Figure 2: The service selection process is comprised of three steps. In the first step the requester sends the policy (either at design time or runtime) to the registry which then returns a list of effective policies from which the ser-



Authorized licensed use limited to: UNIVERSITAETSBIBL STUTTGART. Downloaded on June 15,2010 at 13:09:23 UTC from IEEE Xplore. Restrictions apply.

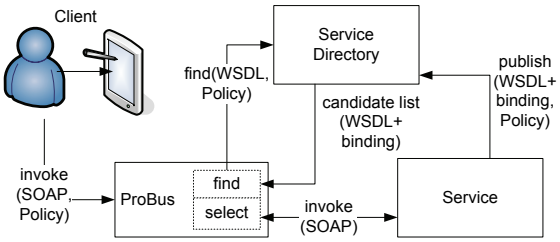


**Figure 3. Class 3: ProBus based dynamic service selection**

vice requester can choose one policy (and therefore the associated endpoint). The requester then sends the actual request to the endpoint that he has chosen from the list. This approach is implemented in standard service middleware, such as Apache Axis 2 [2]. It is used in enterprise contexts [4], [14] [20], as well as Grid scenarios. Using WS-Policy to define mainly security capabilities and requirements in Grids has been proposed in [7] [15] and in [13] for scientific Workflows. This has following disadvantages: Additional functional components are needed on the client side (i.e., to perform the "find" and "select" operations). Since the approach comprises the overhead of requesting a list from the server, transmitting the list to the client and then selecting one of the candidates, it is typically only done once at deployment time, and then always the same service is used.

**Class 3: The policy aware service and resource bus** The contribution of this paper is to introduce an approach for dynamic service binding that is as easy to use as manual binding (class 1) but is as powerful as the autonomous dynamic service binding (class 2). This is accomplished by the extension of an existing service bus (Apache ServiceMix). The extended service bus ProBus is capable of providing a *one-step WS-policy based service selection* as shown in Figure 3. One-step means that only one message is sufficient for selecting and using a service. In case ProBus does not find a suitable service, a corresponding error is returned. ProBus has the advantage that it is very easy to use - the client always calls the same interface on the ProBus and delegates the selection completely to the bus by specifying the policy together with the invocation request in one message. Therefore no additional functional components (to perform the find and select operations) are needed on the client.

Listing 1 shows how a policy can be included in the header of a SOAP message[1]. The element `requiredPolicy` contains the policy the service

[1]The namespace prefixes for all examples are the following: `soap` refers to the SOAP namespace, `wsp` refers to the WS-Policy namespace, `rb` refers to http://www.servicemix.org/policy, `prn` refers to http://www.example.org/printer

```
<soap:Envelope ...>
 <soap:Header>
  <rb:requiredPolicy>
   <wsp:Policy>
    <wsp:ExactlyOne>
     <wsp:All>
      <prn:printer prn:maxCostPerPage="$0.3"/>
     </wsp:All>
    </wsp:ExactlyOne>
   </wsp:Policy>
  </rb:requiredPolicy>
  </soap:Header>
 <soap:Body>
  <!-- Payload of the message -->
 </soap:Body>
</soap:Envelope>
```

**Listing 1. SOAP message with policy included in the header**

client expects the service to follow (or expects ProBus to select a service that is compatible with it).

In this class we explicitly need to tackle the domain-specific post-processing issues in the bus. Therefore we define a general way how to define these domain-specific post-processing rules. As WS-Policy is rendered as XML, we choose XPath to describe these rules. Therefor the mechanism allows to deploy rules that for example state that the `maxCostPerPage` attribute of the requester policy must be smaller than the value of the `maxCostPerPage` attribute of the provider policy.

## 3. Scenario for Dynamic Resource Selection

A resource represents a stateful service. In common understanding a *service is invoked*, such an invocation always produces the same results for the same input values. By that a *service is stateless* which is an important quality because it allows services to be implemented scalable by virtualizing them. In contrast in this paper we define the *interaction with a resource as usage*. The difference between use and invoke is that usage changes the state of a resource. That means a *resource is stateful* which has the implication that each time the resource is used the reaction could be different even for the same input values. So how a resource reacts and what it is capable of is dependent on its actual state. This state is described in a resource property document. That document changes frequently, which demand a very dynamic binding of resources. For every usage of a resource the binding has to be reviewed. In contrast a service has to be bound only once and can be invoked afterwards as



Authorized licensed use limited to: UNIVERSITAETSBIBL STUTTGART. Downloaded on June 15,2010 at 13:09:23 UTC from IEEE Xplore. Restrictions apply.

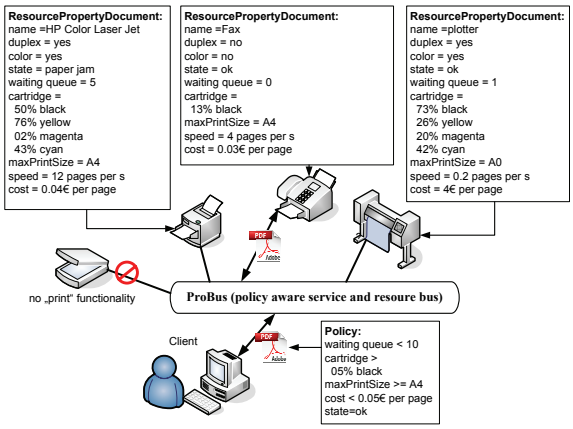


**Figure 4. Printer scenario for dynamic resource selection**

long as it is available. For a better understanding of that following simplified scenario is given.

In a company several printers are available. A printer can be seen as a resource with a WS-Resource interface providing a resource property document as shown in Figure 4. Normally a user has to select the printer manually. For that the user checks the state and properties of each printer and selects the best fitting one to use. This has to be done for every print job again, because the state of the printer could have changed in between, e.g. from (state=ok) to (state=paper-jam). Resource selection is even dependant of the kind of document the user wants to print, e.g. if it is a black and white document or a colored document. That shows how important dynamic binding of resources is.

Because of that we implemented ProBus that provides dynamic resource and service selection based on policies in a *one-step interaction*. Figure 4 shows how the printer scenario is supported by ProBus: The client wants to print a black and white document. To do that he describes his requirements for the printing as a policy. Then he sends the document together with the policy to ProBus. The bus then executes the find and select functionality and routes the document to a resource that provides the printing functionality and fulfills the requirements. As return value the client gets the identifier of the used resource.

## 4. Using ProBus for WS-Policy based Resource Selection

The Web service Resource Framework (WSRF) [8] is the standard to render stateful resources as Web services. It consists of several specifications that specify different aspects of stateful resources (so-called WS-Resources). The WS-ResourceProperties specification [9] standardizes how the state is rendered for WS-Resources. Therefore it introduces a so-called *resource properties* document. It contains a set of resource properties in terms of XML elements. These resource properties can be accessed and manipulated via a set of standard operations such as: `GetResourceProperty` to retrieve a resource property; `SetResourceProperties` to manipulate a resource property; `GetMultiple-ResourceProperties` to retrieve several resource properties at once; or `GetResourcePropertyDocument` to retrieve the whole resource property document at once.

Similar to a WS-Policy that describes non-functional properties of a Web service, a resource properties document describes functional and/or non-functional properties of a WS-Resource. The mechanisms described above help to select a particular service from a group of functionally equivalent services. The same mechanism can be employed to select a particular resource from a pool of resources from the same type. We therefore detail the example from Section 3 into the following: A set of resources that provide printing services can be accessed via a management Web service. These resources expose resource property documents, which contain a resource property that describes their state, the cost per page and the size of their printing queue. A client wants to print a document on these printing resources (but it does not care on which one). Since the printing should happen quickly and must be cheap only cheap printing resources with a small printing queue should be selected. The client could now query all printing resources for their price and queue size by invoking their `GetResourceProperty` operations. This implies that the client knows all ´printing resources and also is able to perform the selection logic which complicates the client development and tightly couples the client to the resources. Therefore we propose a different approach that employs “make-it-happen” semantics for such requests. Using this approach, the client simply sends a printing request that contains a header with a WS-Policy that specifies the maximum cost and queue size. The middleware (ProBus) then selects the appropriate resource based on the published resource properties document and forwards the request to the resource. Therefore the client does not need to know which resources exist and is completely decoupled from them.

This approach however entails several problems, which have been identified and are tackled in the next sections:

1. How does the client know which particular resource stored his data so that he can retrieve it later? In



Authorized licensed use limited to: UNIVERSITAETSBIBL STUTTGART. Downloaded on June 15,2010 at 13:09:23 UTC from IEEE Xplore. Restrictions apply.

more general terms: How are conversations between client and stateful resource handled if the client does not know the concrete Endpoint Reference (EPR) of the resource? (see 4.1)

2. How can a client define policies on resource properties? (see 4.2)
3. How does the bus select concrete resources based on these policies? (see 4.3)
4. How does the bus handle the dynamic appearance and disappearance of resources? (see 4.4)
5. How does the bus handle exceptions, i.e., when no suitable resources can be discovered? (see 4.5)

### 4.1. Handling Conversations

When simply sending a policy with the request the client does not know which concrete resource dealt with his request. This imposes a problem in case several messages need to be exchanged in a complex conversation with this particular resource. Two possible solutions shown in Figure 5 come to mind when trying to solve this problem.

**EPR based solution:** The bus replies to the request with a message that contains the EPR of the resource. In case of a synchronous request-response message exchange the bus can also attach this information to the response message of the resource.

**ID based solution:** The client inserts a correlation token uniquely identifying a conversation in the message, e.g. using WS-Addressing [16] *reference parameters*. The bus then keeps track of which resource is responsible for which conversation by storing a (conversation id, EPR) map. Subsequent requests with a conversation id are then forwarded to the associated resource.

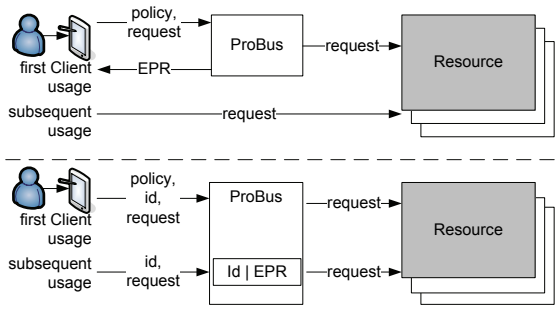


**Figure 5. Two possible solutions to handle conversations with dynamically selected resources**

```
<rb:resourcePropertyCondition
    rb:type="xs:QName">
  <rb:XPath rb:propertyQName="xs:QName">
    xs:string
  </rb:XPath>+
<rb:resourcePropertyCondition>
```

**Listing 2. Pseudo-Schema for the resource property condition assertion type**

```
<prn:Printer>
  <prn:waitingQueue>10</prn:waitingQueue>
  <prn:costPerPage>0.03</prn:costPerPage>
  <!--other resource properties -->
</prn:Printer>
```

**Listing 3. Example resource properties document for printer resource**

### 4.2. Defining Policies Relating to Resource Properties

Similarly to the approach described above, a mechanism must exist to describe how the requester policies are evaluated against the resource properties documents of the resources. We therefore introduce a new WS-Policy assertion type, the *resource property condition*. It allows defining assertions to be evaluated against resource properties. The pseudo-schema of this assertion type is shown in Listing 2.

A *resource property condition* assertion type contains a `rb:type` attribute that refers to the QName of the resource property document. One or more `XPath` rules describe how the post-processing of individual resource properties within this document is handled. All XPath conditions inside of `rb:XPath` elements must evaluate to true in order that the assertion is compatible with a certain resource property document. The logical AND and OR operators of XPath can be used inside an `rb:XPath` element to formulate more complex conditions. The `rb:propertyQName` attribute of an `rb:XPath` element denotes for which resource property this rule applies. A printer resource exposes for example the resource properties document shown in Listing 3.

In order to only send the request to a printer resource that has a waiting queue smaller than 10 the client includes a policy in the request that contains a *resource property condition assertion* as shown in Listing 4.



Authorized licensed use limited to: UNIVERSITAETSBIBL STUTTGART. Downloaded on June 15,2010 at 13:09:23 UTC from IEEE Xplore. Restrictions apply.

```
<wsp:Policy ...>
  <wsp:ExactlyOne>
    <wsp:All>
      <!-- some assertions -->
      <rb:resourcePropertyCondition
          rb:type="prn:printer">
        <rb:XPath
            rb:propertyQName="prn:waitingQueue">
          number(prn:waitingQueue/text()) <10
        </rb:XPath>
      </rb:resourcePropertyCondition>
      <!-- more assertions -->
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

**Listing 4. Policy containing a resource property condition assertion**

### 4.3. Resource Selection Based on Policies

Upon receipt of a message including a header with a policy as shown in Listing 1, the policy processor must follow the following algorithm:

1. Evaluate the policy against the provider policy associated with the Web service that handles the resource manipulations.
2. In case one or more non-empty alternatives have been found, the computed effective policy is handed to the post-processing as described in Section 2.1.
3. Once the post-processor detects two `rb:resourcePropertyCondition` elements in one of the alternatives of the effective policy, the handling is handed over to the *WSRFAssertionProcessor*. The WSRFAssertionProcessor then performs the following steps until a suitable resource is found:

   First it performs a `GetResourceProperty` request on the next resource of the requested type. The request contains the value of `rb:type` attribute of the requester assertion to indicate which resource property is requested. Then it evaluates the result of the `GetResourceProperty` request.

To prevent multiple `GetResourceProperty` requests per resource the requester policy can include a `rb:resourcePropertyDocumentCondition` assertion. This assertion in turn includes one or more `rb:XPath` elements that define rules directly on the resource document. The WSRFAssertionProcessor then only needs to perform one invocation of the resource via the `GetResourcePropertyDocument` operation to retrieve the whole resource properties document. This approach has a drawback as the `GetResourcePropertyDocument` operation is optional and therefore can be only used with resource types that implement this operation. Again, the provider policy must include the resource property document condition assertion to indicate that this assertion can be used and that the corresponding operation is available.

### 4.4. Dynamic Resource Management

In dynamic settings such as Cloud Computing and Grid environments resources are created and destroyed frequently as they appear and disappear dynamically. To select resources dynamically based on resource property (document) assertions, the bus must know which resources have been created. In order to obtain this knowledge, several approaches are possible:

After creation the new resource registers with the bus. However, this requires that the resource knows about the bus and is capable to register itself there. This mostly implies modifications on the resource's implementation. A more suitable approach allows defining which operations on endpoints accessed via the bus act as resource factory and create new resources. The middleware can then evaluate the response of the operation to determine the endpoint reference (EPR) of the resource that can then be used for `GetResourceProperty` requests later on. This approach has been implemented in our ProBus prototype.

Regarding the destruction of resources our prototype implements a lazy approach. That means that the bus only knows about destroyed resources after it has received a `resourceUnvailable` exception after invoking an operation (such as `GetResourceProperty`) on a resource. In this case it removes the resource from its internal resource registry. A different approach would be to notify the bus upon deletion of resources. However, this would require that the resource notifies the bus upon its deletion which is not always feasible as the implementation of the resource might need to be changed.

### 4.5. Exception Handling

When regarding exceptions in service oriented systems, two main kinds of exceptions need to be distinguished: Exceptions on the application level and on the middleware level. Application level exceptions occur during the interaction with a resource and are thrown because of some internal computation error. ProBus simply forwards this exception to the client. In some scenarios it might be appropriate to automatically resend



Authorized licensed use limited to: UNIVERSITAETSBIBL STUTTGART. Downloaded on June 15,2010 at 13:09:23 UTC from IEEE Xplore. Restrictions apply.

the request to another suitable resource. However, this is out of scope of this paper and subject of future work.

Another area where exceptions occur is the middleware level. Normal middleware exceptions include failures such as connection failures. In this area ProBus introduces a new exception type, the `noSuitableResourceFound` exception. This exception extends the faults defined in the WS-Resource specification [8]. The `noSuitableResourceFound` exception is thrown by ProBus if ProBus is unable to find a resource that matches the requester policy. In addition a future extension to ProBus will return a list of effective policies for the available resources which is ranked according to their proximity to the requester policy. By that the client can adapt its policy so that the next request is succesful and does not need to employ a trial and error approach.

## 5. Architecture & Implementation

A suitable architectural foundation for our approach is offered by Enterprise Service Buses (ESBs). ESBs provide means for endpoint virtualization, i.e. service consumers still have to know about the service interface but not about the actual implementation. They bind against so called virtual endpoints and the bus takes care about the routing to a particular instance of a partner. For our prototype, we decided to extend Apache ServiceMix as it provides both, endpoint virtualization and a sophisticated deployment model so that, after making endpoint routing policy-aware, policy related artifacts can be easily deployed.

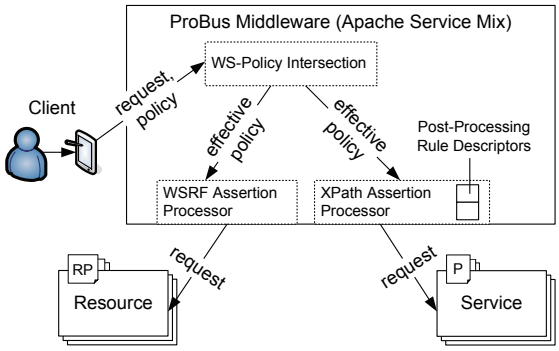


**Figure 6. ProBus dynamically selecting services and resources based on policies and resource properties**

The extensions we have made to ServiceMix are currently being prepared for a code donation to Apache. Our prototype allows registering policies for services. Users can deploy post-processing rules described in post-processing rule descriptors to handle complex assertions. Using resource factory descriptors users can mark operations of services as resource factory operations which create resources that then are automatically registered with ProBus. Clients can then include their requirements in the form of resource property assertions in the header of their SOAP messages that the middleware evaluates against the resource property documents of the resources to select an appropriate resource for the request. As shown in Figure 6 we implemented post-processors for standard WS-Policy assertions using post processing rule descriptors. Another implemented post-processor is the one that evaluates assertions against WS-Resource properties. Additional post-processors (e.g. to delegate resource selection to sophisticated resource management systems) can be plugged in.

## 6. Comparison to Related Work

Virtualization of services is an essential element in service oriented architecture. WS-Policy is a broadly accepted way [17] to describe non-functional properties of services. The traditional approach for policy-based service selection at design time and runtime is implemented in enterprise settings as well as Grid environments, mainly to advertise and select services based on their security policies (see Section 2).

The Grid deals with the virtualization of resources, e.g. through resource brokers [5, 21, 3]. Grid users can send for instance job submissions to the broker. The broker then decides which resources will perform the job. Therefore these resource brokers are similar to our approach where a service/resource consumer sends an invocation message that is then distributed to the right service/resource. Our approach differs from existing resource brokers and ESBs as it does not distinguish between resources and services and is not focused on either one. Compared to Grid resource brokers our ProBus middleware employs a concept (WS-Policy) well accepted in the enterprise domain. Our concept differs from service level agreement (SLA) management approaches (such as WSLA [6] or WS-Agreement [1]) as it has a different focus that is based on the selection of a set of predefined resources. SLAs can be negotiated independently of ProBus with each of the providers of such resources. Several standards exist based on WS-Policy, such as WS-SecurityPolicy [12] to define security assertions based on WS-Security; WS-Reliable Messaging Policy Assertion [11] to define reliable messaging assertions. Other WS-* standards such as WS-Coordination, WS-AtomicTransaction, WS-BusinessActivity [10] integrate natively with WS-Policy and can thus be exploited by ProBus. Our approach therefore brings a standards-based "make-it-happen" approach to service selection



Authorized licensed use limited to: UNIVERSITAETSBIBL STUTTGART. Downloaded on June 15,2010 at 13:09:23 UTC from IEEE Xplore. Restrictions apply.

in the enterprise as well as the Grid. Additionally the presented approach does not rely on any specific (Grid) infrastructure and is extensible to incorporate new standards based on WS-Policy. Our approach is integrated in the ESB. The ESB concept is a fundamental concept in service oriented architecture and is therefore often implemented through a dedicated piece of middleware such as Apache ServiceMix. Therefore our approach eliminates the need for additional middleware and can be integrated with other ESB functionality such as message routing or message transformation.

## 7. Conclusion and Outlook

In this paper we presented an approach how to optimize policy-based service selection in dynamic environments. We introduced the notion of one-step service selection as a means that completely decouples clients and service providers, as service selection is pushed completely into the bus. We showed how requester policies can be embedded in request messages that are then evaluated dynamically. In addition we described a mechanism to deploy post-processing rules These post-processing rule descriptors are based on XPath and allow domain experts to specify very complex domain-specific post-processing rules.

We introduced an approach how service requesters can formulate WS-Policy assertions that evaluate against resource properties of WS-Resources. This allows for policy-based selection of resources without modifying existing WSRF resource implementations. Our one-step service selection approach works best in dynamic settings where resources and services possess dynamic policies and resource properties. In these settings where policies need to be evaluated on every request our approach for one-step service and resource selection based on WS-Policy reduces the amount of message exchanges between requester and ESB middleware. In future work we will combine ProBus with a BPEL engine to define complex dynamic policy-based resource provisioning workflows based on a standard SOA infrastructure. We will also investigate how failover mechanisms can be implemented, that deal with errors in resources by resending the request to a different suitable resource.

## References


[1] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu. Web Services Agreement Specification (WS-Agreement) Version 2005/09. In *Global Grid Forum*, 2005.

[2] Apache Software Foundation. Apache Axis 2. http://ws.apache.org/axis2/, 2007.

[3] R. Buyya, D. Abramson, and J. Giddy. Nimrod-G Resource Broker for Service-Oriented Grid Computing. *IEEE Distributed Systems Online*, 2(7), 2001.

[4] S. Chaari, Y. Badr, and F. Biennier. Enhancing web service selection by QoS-based ontology and WS-policy. In *SAC '08*, 2008.

[5] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2004.

[6] A. Keller and H. Ludwig. The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. *Journal of Network and Systems Management*, 11(1):57–81, 2003.

[7] N. Nagaratnam, P. Janson, J. Dayka, A. Nadalin, F. Siebenlist, V. Welch, I. Foster, and S. Tuecke. Security Architecture for Open Grid Services. *Global Grid Forum Working Draft. Revision as of*, 6(5), 2003.

[8] OASIS. Web Services Resource 1.2 , 2006.

[9] OASIS. Web Services Resource Properties 1.2 , 2006.

[10] OASIS. WS-TX 1.1 OASIS Standards, 2007.

[11] OASIS. Web Services Reliable Messaging Policy Assertion 1.2, 2008.

[12] OASIS. WS-SecurityPolicy 1.2, 2008.

[13] S. Perera and D. Gannon. Enabling Web Service Extensions for Scientific Workflows. *HPDC2006 (WORKS06)*, 2006.

[14] T. Phan, J. Han, J.-G. Schneider, T. Ebringer, and T. Rogers. Policy-Based Service Registration and Discovery. In R. Meersman and Z. Tari, editors, *OTM Conferences (1)*, volume 4803 of *Lecture Notes in Computer Science*, pages 417–426. Springer, 2007.

[15] S. Singh and S. Bawa. A Framework for Handling Security Problems in Grid Environment using Web Services Security Specifications. *Conference on Semantics, Knowledge, and Grid*, 2006.

[16] W3C. Web Services Addressing 1.0 - Core, W3C Recommendation, 2006.

[17] W3C. Testimonials for WS-Policy 1.5. http://www.w3.org/2007/07/wspolicy-testimonial, 2007.

[18] W3C. Web Services Policy 1.5 - Attachment, W3C Recommendation, 2007.

[19] W3C. Web Services Policy 1.5 - Framework, W3C Recommendation, 2007.

[20] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More*. Prentice Hall PTR Upper Saddle River, NJ, USA, 2005.

[21] C. Yang, P. Shih, and K. Li. A high-performance computational resource broker for grid computing environments. *AINA 2005.*, 2005.




Authorized licensed use limited to: UNIVERSITAETSBIBL STUTTGART. Downloaded on June 15,2010 at 13:09:23 UTC from IEEE Xplore. Restrictions apply.