Policy-based Service Registration and Discovery*

Tan Phan[1], Jun Han[1], Jean-Guy Schneider[1], Tim Ebringer[2], Tony Rogers[2]

[1] Faculty of ICT, Swinburne University of Technology, 3122 Hawthorn, Australia
{tphan, jhan, jschneider}@ict.swin.edu.au
[2] CA Labs, CA (Pacific), Building 10, Level 2, 658 Church Street, 3121 Richmond, Australia
{Tim.Ebringer, Tony.Rogers}@ca.com

Abstract.

The WS-Policy framework has been introduced to allow policy to be expressed and associated with Web Services thereby enabling organizations to manage the quality of their services. How the specified polices are kept consistent with the organization's regulations, and how to match service and client policies requirements for effective service discovery, are issues yet to be addressed. In this paper, we present a new approach that allows for the automatic verification and matching of policies, using a service registry that serves as a policy storage and management facility, a policy checkpoint during service publication and as a policy matchmaker during service discovery. We extend WS-Policy with a policy conformance operator for policy verification and use WS-Policy Intersection for policy matching. We develop a policy information model and policy processing logics for the registry. An implementation of a policy-enabled service registry is also introduced.

1. Introduction

Requirements for quality and standard compliance are often specified in the form of policies about the non-functional requirements of various components in an organization's IT infrastructure. In the Web Services context, the WS-Policy [1] framework provides a way to describe the policies regarding Web Services in a machine readable form; this allows for automatic policy enforcement via policy-aware clients. At present, policies are associated with Web Services in various ways and there is no automatic mechanism to guarantee that the policies specified are consistent with the organization or application's specific requirements. There is, therefore, a need for reliable techniques to evaluate policies on a large number of services and a final check point for policy conformance before the services are published and made available to the clients.

Another concern is service discovery which, at present, focuses more on functional and less on non-functional aspects of the services. WS-Policy allows a service to advertise its capabilities and specify its requirements. Unfortunately, there is no easy way for the client to have access to the policy information; this typically requires some

* This work is supported by the Australian Research Council and CA Labs.

out of band communication. Other approaches such as WS-MetadataExchange [4] have suggested ways of adding service meta-data such as policy information into service endpoint description but this still requires the client to know the service endpoint's address.

In this paper, we present an approach to address these issues using a service registry that holds policy information. We argue that a verification unit should reside inside the service registry to verify the services for policy conformance when they are published. We also propose that service clients should be able to indicate their policy requirements as part of the service query. We thus present a management model and techniques to enable automatic policy verification and matching. A prototype tool has also been implemented to demonstrate the approach.

2. Background

The WS-Policy framework comprises a set of specifications that together offer "mechanisms to represent the capabilities and requirements of Web Services as policies" [15]. The framework includes the WS-Policy language [4], which provides a simple and extensible notation to combine the various kinds of policy assertions and form policy descriptions, and the WS-PolicyAttachment specification [3] which specifies how to associate a policy with Web Services entities (services, endpoints, operations, and messages). Various domain-specific standards have also been defined to allow for the expression of policy assertions in individual domains like WS-SecurityPolicy [12], WS-ReliableMessagingPolicy [5], and MTOM [10]. Policy-aware tools such as WSE [13] can generate code to perform policy enforcements automatically. WS-Policy is by itself a simple declarative language with the following normalized structure.

```
<wsp:Policy> <wsp:ExactlyOne>
  (<wsp:All>  (<Assertion>  …  </Assertion>)*  /wsp:All>)*
</wsp:ExactlyOne> </wsp:Policy>
```

Effectively, in its normal form a WS-Policy policy is a logical XOR of the contained policy alternatives with each policy alternative being a logical AND of the assertions contained as seen in the following expression:

$P = XOR(AND(A_{11},...,A_{1m_1}),...,AND(A_{n1},...,A_{nm_n}))$: Where P is a policy expression; A_{ij} ($0 \le i \le n, 0 \le j \le m$) is the jth policy assertion in the ith policy alternative of the expression. Any WS-Policy policy expressions can be normalized into the above form. Therefore, for the sake of simplicity, and without losing generality, in this paper we treat all WS-Policy policy expressions as if they are in their normal form.

A service registry holds service metadata for the registration and discovery of Web services. Registries are characterized by rich metadata management and rich query capabilities. There are two popular registry specifications: UDDI [7] and EbXML Registry [9]. The two specifications were originally created for standardizing inter-organizational service registry products. They are now adopted more for intra-organizational registries due to the trust and privacy issues related to service registration and discovery spanning multiple organizations. At present, neither UDDI nor EbXML Registry has direct support for policy processing.

3. Enabling Policy-based Service Registration and Discovery

We advocate the use of a service registry to support policy verification and policy-based discovery. A service registry is where all service metadata is registered and stored making it suitable for capturing and processing policy information. Existing service metadata management mechanisms in the registry can be leveraged to support the creation, publication, modification and removal of policy.

In our approach, a typical service registry's data model is enhanced with a policy information model to represent policy information. Two additional units, the `PolicyValidator` for service policy verification at publishing time and the `PolicyEnabledQueryManager` for policy matching at the service discovery time, are added to the registry as can be seen in Figure 1.

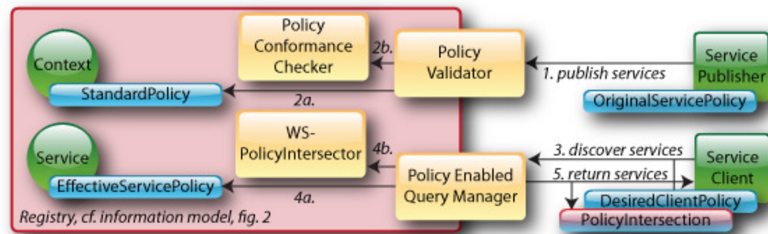

Fig. 1. A registry-based model for policy registration and discovery

Contexts are defined within a registry to represent organizations, applications or development projects. Policy can be specified per context to represent all the common requirements, such as those about security like authentication, authorization, message encryption and signing, that any entity under that context must follow. When a service is published into a context with a policy, that policy must be verified against the policy requirements of the context. The service will only be stored in the registry when the policy conforms to the requirements. When the service policy or context policy is updated, policy will be revalidated. This guarantees that only the services with appropriate policies are made available for consumption by clients.

When a service client looks for a service in a registry, the client may only be interested in a set of services that, apart from satisfying the functional requirements, also support certain policy requirements. Client side policy requirements are conveyed by sending the registry a description of the desirable policy as part of the selection criteria of the service discovery. Only services that support the desired client policies are returned. With the use of a policy-aware registry non-functional (policy-based) and functional discovery can be achieved at the same time.

3.1 Policy Information Model

To support the storage and matching of service policy information inside the registry, we use the abstract information model presented in Figure 2.

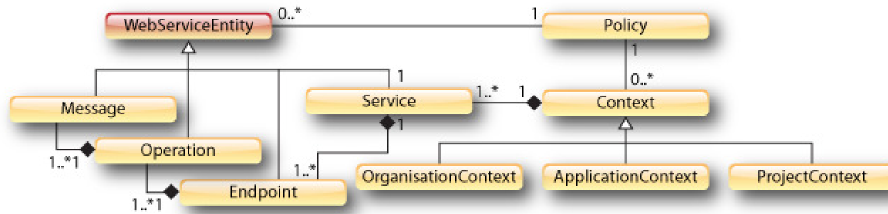

Fig. 2. UML Class diagram for policy information model

This model depicts the relationship between Web Services entities, the context that the entities are deployed to, and the policies that are specified for the entities and the context. Web Services entities are modeled following WSPolicy-Attachment [3]. Essentially, a service consists of one or many physical endpoints, with each endpoint consisting of a collection of operations, and each operation in turns containing a collection of messages. They are all referred to as `WebServicesEntity`.

`Context` models an aggregation of `RegistryEntities` with some common settings and typically represents an organization, an application, or a development project. A Service must exist under one context, which is also the context of the service's endpoints, operations, and messages. A context is associated with one policy description (called the `StandardPolicy`) which encapsulates all the policy requirements that objects to be deployed into the context must conform to. Policy represents a policy description, which might be the (WS-Policy) *merge* [4] of multiple (WS-Policy) policy documents. The association between a Web Services entity and a policy is the association between the entity itself and its *effective* policy. The *effective* policy of a given Web Services entity is a *merge* of the policy that is attached to the entity itself and any policies that the entity inherited from its container entities, following the mechanism specified in WS-PolicyAttachment [3].

3.2 Enabling Policy-based Service Registration

A service published into the registry must have an associated context which is indicated in the service publishing message. Context policy represents policy requirements that apply to *every* entity deployed under the context. The service provider might have different policy requirements for each of the endpoints, operations or messages in the service he publishes so different policies might be specified for the services, endpoints, operations, and messages themselves. In this case, the *effective* policy of each of these entities must conform to the *`StandardPolicy`* for the service to be considered conforming to the requirements of the context.

Policy conformance: Policy conformance verification has a non-commutative nature. That is, the fact that a policy P_1 conforms to another policy P_2 does not imply that P_2 conforms to P_1. Currently, WS-Policy Intersection [4] is referred to as the WS-Policy's operator for policy conformance checking [15]. Being a commutative operator, WS-Policy-Intersection is unsuitable for the checking. Below, we propose a WS-Policy

Conformance operator which is non-commutative and asymmetric. We first start with a general definition for policy conformance

Definition 1: *A policy P_1 is said to conform to a policy P_2 when the fact that one entity satisfies P_1 implies that the entity also satisfies P_2.* This implies that P_1 specifies equal or more stringent requirements and/or equal or more capabilities than P_2 does.

Policy conformance for WS-Policy: Because, in normal form, a WS-Policy policy is an XOR of different policy alternatives, Definition 1 can then be refined as follows.

Definition 2: *In their normal form, a WS-Policy expression P_1 is said to conform to another WS-Policy expression P_2 when the fact that an entity satisfies/supports an alternative in P_1 implies that the entity satisfies/supports one alternative in P_2.*

Definition 3: *A WS-Policy policy alternative PA_1 is said to conform to another policy alternative PA_2 when the fact that one entity satisfies PA_1 implies that the entity also satisfies PA_2.* A WS-Policy alternative is a logical AND of policy assertions, therefore the support for *any assertion* in PA_2 above is implied by the support of *all the assertions* in PA_1. At the assertion level, we rely on a domain-specific conformance function to determine whether a set of policy assertions collectively satisfy another assertion.

The following is an algorithm for verifying policy conformance in WS-Policy derived from the definitions given above. The algorithm assumes the presence of a domain-specific conformance function for any assertions within each of the policy. In the absence of the domain-specific conformance function, the default rule, which is *an assertion conforms to and only to itself*, is applied.

Policy conformance algorithm: Given two normalized policies P_1, the policy that needs to be verified, and P_2, the standard policy:

1. If P_1 is empty (P_1 has no alternative, meaning the set of behaviors accepted by P_1 is empty) then P_1 conforms to P_2 regardless of P_2's structure
2. If P_2 is empty then P_1 does not conform to P_2 unless P_1 is also empty
3. P_1 conforms to P_2 when *each* alternative in P_1 conforms to one alternative in P_2.
4. Given a policy alternative A_1 of P_1 and a policy alternative A_2 of P_2
   a. If A_2 is empty (A_2 has no assertion meaning it can accept any behavior) then A_1 always conforms to A_2
   b. If A_1 is empty then A_1 does not conform to A_2 unless A_2 is also empty
   c. A_1 conforms to A_2 when *each* policy assertion in A_2 is satisfied by *all assertions* in A_1 collectively, using a predefined domain-specific conformance function

Domain-specific conformance function: We assume that policy assertion authors (such as the WS-SecurityPolicy committee members) supply, together with the assertions, the domain-specific policy conformance function. The conformance function for a domain should be able to determine, within the domain, whether a logical AND of a set of assertions conforms to an arbitrary assertion; this is required because sometimes an individual assertion might not conform to another one while a logical AND of assertions might. For example, if we have a policy assertion S_1 requiring the encryption of the entire SOAP message (i.e. encrypting the *envelop* element), an assertion S_2 requiring *header* encryption, and an assertion S_3 requiring *body* encryption, then neither S_2 nor S_3 alone conforms to S_1, but S_2 and S_3 collectively would conform to S_1.

Policy verification processing: The `PolicyValidator` unit of the registry extracts the service's context from the service publication message and, based on the context, retrieves the context's *`StandardPolicy`*. It also extracts the service's

attached *OriginalServicePolicy* and forwards the two policies to the PolicyConformanceChecker for verification using the presented algorithm.

In case separate policies are specified for endpoints, operations, and messages, the registry will first calculate the *effective* policies for each of the entities. The calculated *effective* policies of the endpoints, operations and messages of the published service are then verified against the *StandardPolicy*. Only when they all conform to *StandardPolicy* is the service allowed to be published under the context.

3.3 Enabling Policy-based Service Discovery

Client side policy requirements are indicated as part of the service discovery query, which can span multiple contexts. Also, the client might have specific policy requirements for service, endpoint, operation, or message levels. In this case, it can supply separate policies and uses the mechanisms specified in WSPolicyAttachment [3] to apply the policies to the endpoint, operation or message scope as needed.

Policy matching is based on the compatibility of the client policy and the service policy. To determine policy compatibility we use the Policy Intersection algorithm defined in WS-Policy. WS-Policy's Policy Intersection is designed to check whether one side of the interaction will support the conditions indicated by the other side and what is the policy that will be mutually manifested on the wire during the interaction. This logic is thus suitable for policy discovery as the client can use WS-Policy Intersection to determine whether a target service supports its own policy requirements. We introduce in brief the policy compatibility logics defined in WS-Policy Intersection.

Policy compatibility: According to Policy Intersection, two policies P_1 and P_2 are said to be compatible when they have a non-empty intersection, meaning that there exists at least an alternative in P_1 that is compatible with an alternative in P_2 and vice versa. A policy alternative A_1 is compatible with a policy alternative A_2 when, for each assertion in A_1, there exists a compatible assertion in A_2 and vice versa. Two policy assertions are said to be compatible with each other if the presence of one implies (in a domain-specific way) the support for the other. In the absence of a domain-specific assertion compatibility function, a policy assertion S_1 is said to be compatible with a policy assertion S_2 when they specifies the same type of capability or requirement (having the same Qualified Name) and if one assertion has a nested policy then the other must also have the same nested policy.

Policy matching processing: For each functionally matched service found, the PolicyEnabledQueryManager resolves the service's associated policy description (the service's *EffectivePolicy*) and forwards it to the PolicyIntersector to perform the intersection. If the intersection policy is non-empty, meaning the service and client policies are compatible, the service is then added to the returned set. In case separate policy requirements are specified for the service, endpoint, operation, and message levels, the registry, upon receiving the client policies, will perform the calculation for the *effective* policies for each level of the client requirements. For each target service, it also calculates the *effective* policies for the service, its endpoint, operations and messages and then performs the policy matching at these levels accordingly. Only when matching is achieved at *all* the specified levels, is the service considered having a policy that matches client's requirements.

3.4 Example

A bank creates the following standard policy for the `CorporateContext` in its internal registry, which requires SOAP *body* encryption for every Web Services message

```
<wsp:Policy…> <wsp:ExactlyOne> <wsp:All> <sp:EncryptedElements>
          <sp:XPath>/S:Envelope/S:Body</sp:XPath>
</sp:EncryptedElements> </wsp:All> </wsp:ExactlyOne> </wsp:Policy>
```

A developer X developed a service named `InterestRateQuote`. He believes the service is critical and thus requires the encryption of the entire SOAP message *envelope*:

```
<wsp:Policy…> <wsp:ExactlyOne> <wsp:All> <sp:EncryptedElements>
          <sp:XPath>/S:Envelope</sp:XPath>
</sp:EncryptedElements> </wsp:All> </wsp:ExactlyOne> </wsp:Policy>
```

For another service `ExchangeRateQuote`, X believes the service can function properly with no further requirements so a policy with only one empty alternative (meaning any behavior is accepted) is supplied:

```
<wsp:Policy…>
        <wsp:ExactlyOne> <wsp:All/> </wsp:ExactlyOne>
 </wsp:Policy>
```

When the two services `InterestRateQuote` and `ExchangeRateQuote` are published into the `CorporateContext` of the registry, only `InterestRateQuote` is accepted because the services' policy, which requires SOAP *envelop* encryption, is stronger than and thus conforms to the standard requirement of SOAP *body* encryption while `ExchangeRateQuote`'s policy requirements is weaker than that of the context.

Another developer Y wants to query the registry to find a live base-interest-rate quote service that supports SOAP *header* encryption as indicated in his desirable policy:

```
<wsp:Policy…> <wsp:ExactlyOne> <wsp:All> <sp:EncryptedElements>
          <sp:XPath>/S:Envelope/S:Header</sp:XPath>
</sp:EncryptedElements> </wsp:All> </wsp:ExactlyOne> </wsp:Policy>
```

When Y submits a query to the registry with this desired policy, the registry returns `InterestRateQuote` because the service's and Y's policy are compatible (SOAP *envelop* encryption implies the support for SOAP *header* encryption and vice versa).

4. Implementation

We have implemented (in Java) the support for policy on top of the CA eTrust UDDI 3.0. We implemented an initial mapping of the policy information model to UDDI version 3 data structures as shown in the following figure.

Fig. 3. Mapping Registry Information Model to UDDI data structures

A high-level architecture of the software is depicted in Figure 4 below. In this figure, the components colored in grey are the added/modified components and the rest are those existing in the current CA eTrust UDDI. Specifically we have modified the

`saveServiceImpl` and `findServiceImpl` components of the registry, which are responsible for registering and discovering services respectively, to accommodate the support for policy processing following the logics described in Section 3. We added the implementation for the policy conformance and WS-Policy intersection algorithm (Section 4) to the open source Apache WS-Policy library – Apache Neethi [1] and use the library as the `PolicyIntersector` and `ConformanceChecker`.

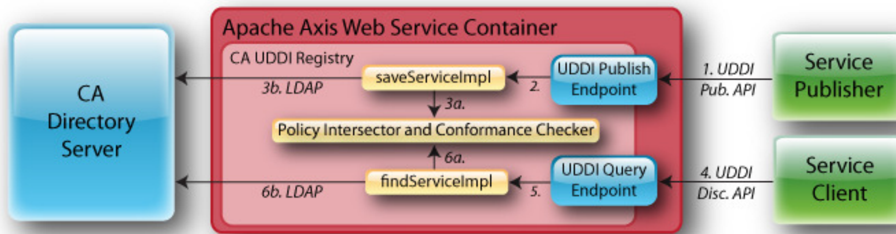

Fig. 4. eTrust UDDI's support for policy

To enable the attachment of policy descriptions to UDDI service publishing and querying messages we define a UDDI *tModel* key for policy information. When the service publisher publishes a service or when the service client queries for a service, they supply the URI to the associated policy description as the value in a key-value pair entry – the UDDI *keyedReference*, with the key of the entry being the predefined policy *tModel,* in the UDDI *categoryBag* of the service.

5. Related Work

There has been a body of work in the area of enhancing service discovery with non-functional matching. UDDIe [14] attempts to extend UDDI with the notion of 'blue pages' for enabling service discovery based on user-defined properties like Quality of Service (QoS) that a service can provide, or the methods available within a service. Similarly, WSLA framework in [11] provides the support for a Service-Level Agreement between a service provider and a potential requestor which allows clients to be able to find services that satisfy their QoS requirements. The main difference between our work and these works is none of the works above discusses how to ensure the properties specified for a service are valid and conforms to the organization's requirements.

The Ponder framework, which includes a specification language, a management model [8] allows for the policy-based management of distributed systems. However, the work focuses more on transport-level access control and QoS management for telecommunication networks rather than for application-level management of SOA systems. Another policy language based on decision tree has been defined in [16] but only limited to specifying QoS parameters for SOA components. This work also advocates the use of a service registry as the storage mechanism for policy information,

but the work does not provide any information on how the policy is to be stored in registry and how the parties involved in policy verification and management can interact with the registry.

6. Conclusion and Future Work

In this paper, we have presented a registry-based framework and the associated techniques to enable the automatic verification of service policy information when a service is registered and the automatic matching of service and client policy information when the service clients query for a service respectively. Verification and match making can be done at the endpoint, operation or message levels of the service. Policy verification is based on a conformance operator we developed for WS-Policy, while policy matching is based on WS-Policy Intersection. In future work, we plan to define a framework for policy-based runtime service and application management using registry.

Reference

1. The Apache Software Foundation (2007). *Apache Neethi 2.0.* Jun, 2007, *http://ws.apache.org/commons/neethi/index.html.*
2. Bajaj S. et al. (2006). *Web Services Policy Framework 1.2.* W3C, Apr, 2006.
3. Bajaj S. et al. (2006). *Web Services Policy Attachment 1.2.* W3C, Apr, 2006.
4. Ballinger K. et al. (2006). *Web Services Metadata Exchange 1.1.* IBM, BEA Systems, Microsoft, SAP, AG, CA, Sun Microsystems, and webMethods, Aug, 2006.
5. Bilorusets R. et al. (2006), *Web Services Reliable Messaging Protocol 1.0.* IBM, BEA Systems, Microsoft, and TIBCO Software, Feb, 2005.
6. Box, D. et al. (2004). *Web services addressing (WS-Addressing).* W3C, Aug, 2004.
7. Clement L. et al. (2004). *Universal Description, Discovery, and Integration 3.0.* OASIS, Oct, 2004.
8. Damianou N. (2002). *A Policy Framework for Management of Distributed Systems*. PhD Thesis, Imperial College, London.
9. Fuger S. et al. (2005). *EbXML Registry Information Model 3.0 and EbXML Registry Service and Protocol 3.0.* OASIS, May, 2005.
10. Gudgin M. et al. (2005). *SOAP Message Optimization Transmission Mechanism 1.0.* W3C, Jan, 2005.
11. Keller A., Ludwig H. (2003). *The WSLA Framework: Specifying and Monitoring Service Level Agreement for Web Services*. J. of Network and Systems Management, 2003.
12. Lawrence K. et al. (2006), *Web Services Security Policy 1.2.* OASIS, 2005.
13. Microsoft (2005). *Web Services Enhancement 3.0.* Released in Jul, 2005, *http://msdn2.microsoft.com/en-us/webservices/aa740663.aspx.*
14. ShaikhAli A., Rana O. F., Ali-Ali R., Walker D.V. (2003). *UDDIe*: *an extended registry for Web Services.* In Proc. Application and Internet Workshop, 2003, Orlando, FL, USA.
15. Vedamuthu A. et al. (2007). *Web Services Policy 1.5 – Primer.* W3C, Jun, 2007
16. Wang C., Wang G., Chen A., Wang H., Pierce Y., Fung C., Uczekaj F. (2005). *A Policy-Based Approach for QoS Specification and Enforcement in Distributed Service-Oriented Architecture.* In Proc. 2005 IEEE Int'l Conf. on Services Computing (SCC'05), FL, USA.