SmartSTEP: automatic configuration of Web Services

João C. C. Leitão[1]

[1] Instituto Superior Técnico, Technical University of Lisbon
Av. Rovisco Pais, 1
1049-001 Lisboa, Portugal
joaoleitao@ist.utl.pt

Abstract: Web Services (WS) are an important tool for the integration of enterprise applications. With a growing set of WS related standards (WS-*), the technology has become increasingly more complicated to configure and manage, even more so when the Quality of Service (QoS) requirements of the system are changing. This paper presents the results of a study conducted on the ability of the major Web Services implementations to adapt to changing QoS attributes. Their shortcomings are then used as motivation for SmartSTEP, a policy-driven automatic configuration solution.

Keywords: Web Services, Quality of Service, WS-Policy, Automatic Configuration, STEP Framework, SmartSTEP.

1. Introduction

Enterprise applications have demanding requirements: many users, large volumes of data, ever-changing business rules, and multiple systems' integration interfaces to connect to other applications [1]. The fundamental challenge is *change* so there is great value in techniques that enable information systems to quickly *adapt* to changes in requirements.

Web Services [2] promise *flexibility* through the usage of *services*, as proposed by Service-Oriented Architecture (SOA) [3]. According to SOA principles, services should present a formal contract (WSDL [4]) with all the information a client requires to use the described service.

A Web Service can be further defined as a network access endpoint to resources: data and business functions [2]. Although this endpoint can be accessed in many different ways, the most common is SOAP[1] [5], an extensible XML-based protocol for exchanging information in distributed environments.

1 Although SOAP was initially defined as Simple Object Access Protocol, the 1.2 version of the standard dropped this definition and simply refers to itself as SOAP.

The three major Web Services implementations are Windows Communication Foundation (WCF) [6], Metro [7] and Axis2 [8].

Recently, these projects have been focusing on the support of WS related standards (WS-*), like WS-Policy [9], a framework for expressing policies that refer to capabilities, requirements or other characteristics of an entity.

Although policies are mainly used for informational purposes, they can also be used as a part of the Web Services configuration process, as shown by some of the studied platforms. Besides the possibility of defining multiple alternatives, the intersection operation is extremely useful in the negotiation of requirements from the actors involved in a remote invocation.

The goal of the SmartSTEP project is to allow the definition and usage of policies as the starting point for the *automatic configuration* of mechanisms that can satisfy them.

The base platform for this system is version 1.3.1 of STEP [10], an academic open-source Java [11] platform with support for Web applications and Web Services. Its main design goals are *extensibility* and *simplicity.*

1.1. Motivation

Section 3. presents a use case that demonstrates how hard it is to configure a mobile and dynamic system. This use case is based in the work of a field salesman. These salesmen need to communicate with their agency's central computer systems as well as other partner agencies' in order to obtain public client data. These central systems present connection requirements like different security levels for connections from different networks.

The difficulty of implementing a system like this led to the SmartSTEP proposal, which can be described as a dynamic and automatic Web Services configuration system.

1.2. Goal

SmartSTEP's goal is to support user-free automatic reconfiguration of message handlers, as a response to an *external event*. A system with the ability to reconfigure itself as a reaction to this type of event is normally referred to as a self-adaptive system [12].

Message handlers are the software blocks responsible for the satisfaction of QoS requirements.

1.3. Contributions

This work has two main contributions:

- The study of the configuration mechanisms of the main Web Services implementations;

- The automatic Web Services configuration system for the STEP platform.

2. Web Services implementations

This section presents the Web Services configuration mechanisms of the major WS implementations: WCF, Metro and Axis2. The original STEP features are also presented.

2.1. Features

The following table summarizes the main configuration features of the studied platforms.

Table 1. Configuration features of existing Web Services implementations

Area	Feature	WCF	Metro	Axis2	STEP
Policies	WS-Policy	Yes	Yes	Yes	No
	Custom policies	Yes (1)	No	Yes	No
	Server-side policy alternatives	No	No	No	No
Configuration	WS-Policy based configuration	No	Yes	Yes	No
	Runtime policy configuration	No	Yes	Yes	No
	Automatic reconfiguration (2)	No	No	No	No
Extensibility	Modular message handlers	No	No	Yes	Yes
	Message handler extensibility	Yes (1)	Yes	Yes	Yes
	Message handler hot deployment	No	No	Yes (3)	No

(1). Requires platform extensions **(2).** Without user intervention
(3). If available in Axis2 Web Application

WCF supports the majority of WS-* standards, but it doesn't support their runtime configuration, making it a very static platform.

Metro fails in the lack of customization. Custom policies are not supported and the difficulty of creating new modules discourages whoever needs to support a new feature.

Axis2 has modules for the main WS-* standards, but there are still many without a public stable implementation.

STEP 1.3.1 doesn't support WS-Policy and its configuration is fully static.

2.2. SmartSTEP requirements

The proposed feature list for the SmartSTEP system is composed by all the features on Table 1, including those unsupported by all studied platforms, namely server-side policies, automatic reconfiguration and handler hot deployment.

The project requirements are listed in Table 2.

Table 2. SmartSTEP requirements

Number	Description
	Justification
SSR1.	The definition of QoS requirements should be done using WS-Policy
	WS-Policy allows the definition of service policies in a standard format easing the future interoperability with other platforms
SSR2.	The platform should support any policy
	This requirement is based on STEP's extensibility principle
SSR3.	The service policies should be defined in its WSDL contrat
	With the definition of policies in WSDL contracts any client can access the QoS requirements of a service
SSR4.	A client should adapt to changes in the policies of a known service
	Required for automatic configuration
SSR5.	The platform should allow runtime modification of policies
	Allows a more dynamic configuration
SSR6.	The platform should allow message handler hot deployment
	Since a service's policies can be modified in runtime and clients may need to adapt to new requirements, it is logical to allow the hot deployment of message handlers in runtime to satisfy the new policies
SSR7.	A service should support policies with multiple alternatives
	This requirement eases the configuration of a service without increasing its functional complexity
SSR8.	The supported policies should be implemented by reusable software elements
	Following in the footsteps of Axis2, this modularization allows the reusage of public implementations
SSR9.	The platform should support the definition of custom configuration mechanisms
	According to STEP's extensibility principle

3. Use case definition

To demonstrate the usefulness of policy-based automatic configuration, a real world use case is defined: insurance sales (*InSales)*.

Let's imagine a field insurance salesman, posted at the local university, selling travel insurances for college students going on their senior trip.

A new client approaches him to create a new account and informs him that he is already registered in a partner agency. After importing the client's data from the partner and registering him as new client, the salesman creates a new insurance proposal. After he explains the proposal details, the client accepts it and the salesman submits the proposal. At the agency's office, a manager studies the proposal and accepts it. After retrieving the result, the salesman informs the client and they discuss any remaining details.

3.1. Requirements

Two different applications are required to implement this use case: central systems (agencies) and mobile systems (salesman and manager).

The connections are made using Web Services, with the mobile systems invoking the services provided by the agencies. The connection and configuration requirements are listed in Table 3.

Table 3. Use case requirements

Number	Description
UCR1.	The connection requirements should be defined using WS-Policy
UCR2.	Policies should be defined using custom assertions
UCR3.	All Web Services should be functionally compatible
UCR4.	A client can invoke any service functionally supported regardless of its endpoint
UCR5.	Clients should adapt to the requirements of any invoked service
UCR6.	The multiple configurations of a Web Service should be defined as policy alternatives
UCR7.	A connection between systems in different networks should use a high security scheme
UCR8.	A connection between systems in the same network should use a low security scheme
UCR9.	The security scheme should be chosen by the client
UCR10.	The client's network should be validated whenever he uses a low security scheme
UCR11.	A service's policies can be modified at any point during the execution of the system

UCR12.	A system can add new message handlers at any point during its execution

3.2. Problems

The studied platforms have some limitations that prevent them from fully supporting the defined requirements. Table 4 presents some problems that their implementation would cause in the studied platforms.

Table 4. Use case implementation problems in studied platforms

Problem			
	WCF	**Metro**	**Axis2**
Definition of custom policies (**UCR2.**)			
	+ Extensions	- Unsupported	+ Modules announce policies
Invocation of any functionally supported service (**UCR3.** and **UCR4.**)			
	- Requires configuration	- Requires configuration	- Requires configuration
Adaptation to the services' requirements (**UCR5.**)			
	- Unsupported	- Unsupported	- Unsupported
Services with policy alternatives (**UCR6.**, **UCR7.** and **UCR8.**)			
	- Unsupported	- Unsupported	- Unsupported
Choice and validation of security schemes (**UCR9.** and **UCR10.**)			
	+ Endpoint choice	+ Endpoint choice	+ Endpoint choice
Runtime modification of policies (**UCR11.**)			
	- Unsupported	+ Management interface	+ Management interface
Message handler runtime extensibility (**UCR12.**)			
	- Unsupported	- Unsupported	- If available in the platform

The SmartSTEP system, described in the next section, intends to solve these problems.

4. SmartSTEP details

This section describes the new features implemented in the STEP platform as part of the SmartSTEP project.

The main focus of this project was the refactoring of the extension system, the support of the WS-Policy standard and the implementation of a library dynamic loading system.

4.1. Extensions

The extensions allow to add new abilities to a system. They can be used to intercept local and remote services to perform additional tasks, like SOAP message processing.

Each extension defines local (*ServiceInterceptors*) and remote interceptors (*WebServiceInterceptors*), executed at specific interception points before and after a service.

In SmartSTEP, each extension announces the policies it can handle, standard or custom (**SSR2.**). WS-Policy interpretation (**SSR1.**) is based on the Apache Neethi library [13].

Since extensions can be used in different applications, they allow to satisfy requirement **SSR8.**.

The extension processing is sequential, where each interceptor has access to the results of the previous interceptor.

In order to allow dynamic configuration, SmartSTEP introduced the concept of *pipe*, which can be described as a list of local or remote interceptors. A pipe is responsible for the ordered execution of its interceptors.

The creation of pipes is achieved through *pipefactories*, SmartSTEP's extension configurators. The factory to use is defined in the `config.properties` file using the property name `extensions.factory`.

Four different configurators were created along the SmartSTEP system:

- **`NullPipeFactory`**: creates a pipe with no interceptors. To use this factory, define the factory property with the value `null`;
- **`PropertiesPipeFactory`**: creates a pipe based on properties from file `extensions.properties`. This is STEP's original extension configuration system. To use this factory, define the factory property with the value `properties`;
- **`PolicyPipeFactory`**: creates a pipe based on WS-Policy (**SSR3.**). Remote service configuration is based on the service's policies and the client policies defined in file `smartstep.xml`[2]. The two policies are intersected and an alternative from the resulting policy is used to activate the respective extensions. To use this factory, define the factory property with the value `policy`;
- **`DynamicPolicyPipeFactory`**: creates a pipe based on WS-Policy. Unlike `PolicyPipeFactory`, this is an adaptive configurator. Instead of intersecting policies, this configurator uses only the service's policy. To use this factory, define the factory property with the value `dynamic`.

2 SmartSTEP's XML-based policy configuration file.

Each application can also use *custom configurators* (**SSR9.**), defined with the classname as value for the factory property.

4.2. JarLoader

The *JarLoader* is SmartSTEP's dynamic loading mechanism. A JarLoader can be created to load all JAR's in given directory. This loading process can be automized to periodically search for new JAR's to install. This mechanism is used to load and install extensions, according to requirement **SSR6.**.

5. Use case implementation

Table 5 presents SmartSTEP's response to the implementation problems identified in section 3.2..

Table 5. SmartSTEP's response to the use case problems

Problem
SmartSTEP
Definition of custom policies
+ Extensions announce policies
Invocation of any functionally supported service
+ Use `StubFactory`[3] to create the service interface for a given endpoint
Adaptation to the services' requirements
+ Using `DynamicPolicyPipeFactory` configurator
Services with policy alternatives
+ Announce the used alternative in a SOAP header
Choice and validation of security schemes
+ Customizing `DynamicPolicyPipeFactory`'s alternative choosing algorithm
Runtime modification of policies
+ Directly in WSDL and `smartstep.xml` files
Message handler runtime extensibility
+ Using the JarLoader

3 Class used to create service interface objects (`Port`). Allows the modification of a service's default endpoint.

6. Evaluation

Table 6 presents the evaluation of SmartSTEP's requirements implementation.

Table 6. SmartSTEP requirements evaluation

Number	Descrição	Implementation
SSR1.	Requirement definition using WS-Policy	**Parcial**
SSR2.	Support for any policy	**Total**
SSR3.	Definition of a service's policies in its WSDL	**Parcial**
SSR4.	Adaptation to changes in a service's policies	**Total**
SSR5.	Runtime policy modification	**Total**
SSR6.	Message handler hot deployment	**Total**
SSR7.	Support for service policy alternatives	**Total**
SSR8.	Support of policies through reusable components	**Total**
SSR9.	Definition of custom configuration mechanisms	**Total**

Because of limitations of the used libraries or for time restrictions, some of the initially planned standards were only partially implemented. Version 1.5 and some elements of version 1.2 of the WS-Policy standard are not supported. WS-PolicyAttachment support is also limited as the definition of policies in the WSDL is restricted to the services.

The full support of these standards requires some implementation effort, but nothing compared to the testing required. This is caused by the complexity of the algorithms defined by these standards. Although not supported, the implementation of the missing elements is already prepared.

7. Conclusion

The study of existing technologies and systems has shown that they focused on the support of multiple WS-* standards instead of simplifying its use.

This led to the proposal of a new system: SmartSTEP. This project's main goal was the creation of an automatic Web Service configuration system based on WS-Policy.

The SmartSTEP system can support virtually any standard, with minimal configuration effort. Human intervention is limited to the definition of policies. The rest of the configuration process is *automatic* and *dynamic.*

Configuration alternatives, *dynamic adaptation* and *runtime extensibility* are some of STEP's new features that make it one of the most powerful Web Services platforms.

References

[1] Hohpe, G., & Woolf, B.: *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Addison-Wesley Professional. (2003).

[2] Alonso, G., Casati, F., Kuno, H., & Machiraju, V.: *Web Services - Concepts, Architectures and Applications.* Springer. (2004).

[3] Erl, T.: *Service-Oriented Architecture (SOA): Concepts, Technology, and Design.* Prentice Hall. (2005).

[4] Chinnici, R., Moreau, J.-J., Ryman, A., & Weerawarana, S.: *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*. (2007). Retirado de http://www.w3.org/TR/wsdl20/

[5] Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J.-J., Nielsen, H. F., Karmarkar, A., et al.: *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*. (2007). Retirado de http://www.w3.org/TR/soap12-part1/

[6] Lowy, J.: *Programming WCF Services, Second Edition.* O'Reilly Media. (2008).

[7] Kalin, M.: *Java Web Services: Up and Running.* O'Reilly Media. (2009).

[8] Tong, K. K.: *Developing Web Services with Apache Axis2.* TipTec Development. (2008).

[9] Vedamuthu, A., Orchard, D., Hirsch, F., Hondo, M., Yendluri, P., Boubez, T., et al.: *Web Services Policy 1.5 - Framework*. (2007). Retirado de http://www.w3.org/TR/ws-policy/

[10] Pardal, M., Fernandes, S. M., Martins, J., & Pardal, J. P.: Customizing Web Services with Extensions in the STEP framework. In *International Journal of Web Services Practices, Vol.3, No.1-2* , 1-11. (2008).

[11] Arnold, K., Gosling, J., & Holmes, D.: *Java(TM) Programming Language, The (4th Edition).* Prentice Hall. (2005).

[12] Heuvel, W.-J. v., Weigand, H., & Hiel, M.: Configurable adapters: the substrate of self-adaptive web services. In *ICEC '07: Proceedings of the ninth international conference on Electronic commerce* (pp. 127-134). Minneapolis, MN, USA: ACM. (2007).

[13] *Apache Neethi*. Retirado de http://ws.apache.org/commons/neethi/