

# Non-Intrusive, Flexible, Role-Based Authorization Extensions for WSRF::Lite

Michael Parkin\*, Bruno Harbulot and John Brooke  
School of Computer Science, The University of Manchester  
Manchester, United Kingdom, M13 9PL

parkinm@cs.man.ac.uk, {bruno.harbulot, john.brooke}@manchester.ac.uk

## Abstract

*This paper describes in detail extensions developed for the WSRF::Lite Grid resource container to provide a flexible and extensible role-based authorization and access-control mechanism. This is achieved by using techniques similar to those used in aspect-oriented programming as it is recognized that authorization is a process that should not be implemented in the main body of code and is a cross-cutting concern. As a result, the solution meets the requirements of being non-prescriptive about the type security token used and, most importantly, ‘non-intrusive’ as the extensions do not require the modification of the existing WSRF::Lite container or application code.*

## 1 Introduction

Authorization is the process of giving an entity permission to access a resource based on a defined access policy. In the context of distributed services, it is necessary to perform authorization decisions on each operation a service offers in order to support role-based access control (RBAC) — acknowledged to simplify the administration of permissions across an organization [7]. Without the ability to provide authorization decisions at the operation level the only option is to grant or deny access to the whole set of operations of the service, a level of authorization which is too coarse-grained; full access and, therefore, full control of the service has to be given to all interested parties even if some of them only require permission to observe the service’s state. Thus, it is essential to restrict access to a resource at the operation level (e.g. read, update, delete, etc.) in order to allow some users read, others update and others delete privileges and to provide an RBAC solution.

Existing methods of achieving this in a Grid environment, such as VOMS [1], are relatively heavyweight solutions that are complex and time-consuming to implement and, when available, are tied to a single type of security tokens; VOMS can only be used with X.509 certificates, for example. Certificates are acknowledged to be difficult to

obtain and use [5] [3] and not all implementations of Grid computing require the level of assurance given by these tokens. This work set out, therefore, to extend existing Grid middleware to define an RBAC authorization capability that can accept many types of authenticated security tokens, and do so in a lightweight and flexible manner.

Our work in the RealityGrid Project [4] uses the WSRF::Lite [13] implementation of the Web Service Resource Framework (WSRF) [10] to provide scientists with access to Grid resources. As will be discussed further in Section 2, WSRF::Lite is a lightweight and highly portable piece of software. However, whilst WSRF::Lite supports the passing of security credentials at the transport level via Transport Layer Security (TLS) and within messages using the WS-Security [12] specification for authentication, the container does not natively support any method of authorization and has the single, basic authorization policy of allowing all authenticated users to access all operations. Thus, a method for providing authorization at the operation level in order to provide RBAC is required for the WSRF::Lite container and is the objective of this work.

As will be discussed in Section 3, authorization is a responsibility that crosscuts several other system concerns. It affects both the container and the resource (i.e. scientist’s application) code, yet does not inherently belong to either. Therefore, authorization can be considered as an *aspect* [11] of the system and there is a need for a flexible implementation of this aspect into the system.

The design goals for implementing this flexible authorization mechanism were three-fold: 1) to be ‘non-intrusive’, i.e. not require modification to either the existing container code or the resource code, necessary as this would lead to a bifurcation of the code base and, as not all users of WSRF::Lite require such an authorization solution, these extensions should not be included in the main body of code; 2) to have the capability to use different authentication tokens and methods of authorization, as not all scientists require the level of security X.509 certificates offer and would like to use more ‘lightweight’ tokens; and 3) to be separated from the concerns of resource provision, which will lead to

a more flexible solution as the authorization decision can be taken in accordance with however the organization to which the requester belongs deems fit, in practice this means the resource asks a policy decision point (PDP) for the authorization decision.

The remainder of this paper is as follows: Section 2 introduces the WSRF::Lite container; Section 3 provides an introduction to Aspect-Oriented Software Development (AOSD); Section 4 describes the authorization architecture for WSRF::Lite; Section 5 provides an example implementation of an RBAC decision process; Section 6 discusses how the techniques and methods used in this work could be applied to other WSRF implementations; and, finally, Section 7 provides a summary and conclusions.

## 2 The WSRF::Lite Container and WS-Resources

As its name suggests, WSRF::Lite is a lightweight, relatively simple Web Service container that can be used to host WS-Resources, which are interacted with over a network using SOAP/HTTP<sup>1</sup>, possibly using mutually authenticated TLS connections. Its advantages over other WSRF implementations, such as the Globus Toolkit [9] and Apache WSRF [2], are that it is extremely lightweight and is supported on most (if not all) platforms as it can be deployed wherever Perl can be installed. WSRF::Lite is mature (it was first released in 2003), stable<sup>2</sup>, highly scalable, supports dynamic service deployment (i.e. new services can be deployed without restarting the container) and, in the event of the container failing, allows the persistence of the WS-Resources it is hosting. Additionally, as Perl is an excellent ‘glue’ language, it is easy to interface the WSRF::Lite container with low-level system software and legacy code written in languages such as C and Fortran. WSRF::Lite has been used extensively (and successfully) for this purpose in the award-winning UK e-Science RealityGrid Project.

When requested by a client via a SOAP/HTTP(S) message, the container uses the factory pattern to create new WS-Resource instances. The WSRF WS-Resource and WS-ResourceLifetime functions are contained in two inheriting Perl classes, as shown in Figure 1, called WSRF::WSRP and WSRF::WSRL, respectively. Thus, in order for scientists to convert their application to a WS-Resource, all that is required is to inherit from the WSRF::RP module. Alternatively, if the developers wish to include WS-ResourceLifetime behavior into their WS-Resource, they can inherit from the WSRF::WSRL module.

<sup>1</sup>The container supports SOAP/HTTP through the Perl SOAP::Lite package available from <http://www.soaplite.com/>.

<sup>2</sup>WSRF::Lite is an Open Middleware Infrastructure Institute (OMII) managed project and is available for download from <http://www.omii.ac.uk/downloads/project.jsp?projectid=71>

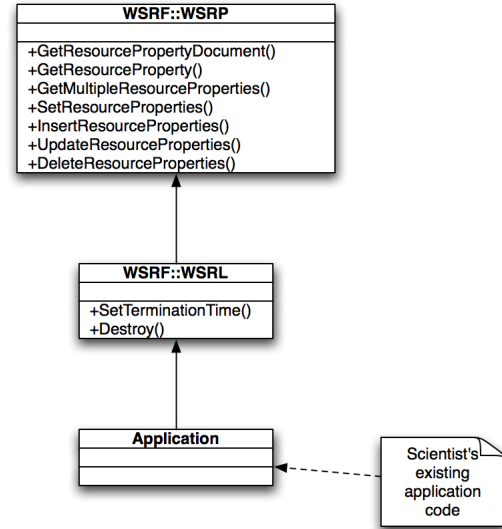


Figure 1. Inheritance of WSRF::Lite Resources

## 3 Aspect-Oriented Programming

A piece of software can be considered as the composition of a set of concerns, often divided into several functional units. In such a decomposition, certain concerns, however, do not entirely belong to any of these units or may affect other units. In this context, two concerns are said to *crosscut* each other when they affect one another without belonging to the same part of the hierarchical decomposition. Typically, their implementation, using procedural, functional or object-oriented languages implies some code-tangling, where the code for a concern is contained in a unit that is mainly aimed at another concern. For example, logging, authentication and authorization are often considered as general crosscutting concerns in that they more or less always crosscut the main purpose of the application.

The goal of Aspect-Oriented Programming (AOP) [11] is to provide a way to represent concerns that crosscut other concerns in a piece of software and provide a unit of decomposition capable of encapsulating *crosscutting concerns* that would otherwise tangle or be scattered in the code. Thus, the general style of programming that arises out of this aim consists of program statements of the form: “*In programs P, whenever condition C arises, perform action A.*” [8]

This leads to the following definitions: a *join point* is a point in a program where a concern crosscutting that part of the program might intervene. Join points are the points that can be used to express potential conditions *C* in programs, according to the above formulation. Join points can be seen as hooks in a program where other program parts can be conditionally attached and executed.

A *pointcut* is a subset of all possible join points. The expression of a pointcut is the *pointcut descriptor* (often, the term “pointcut” is used in place of “pointcut descriptor”). A pointcut descriptor defines the condition  $C$  in the above formulation. This condition matches a subset of join points which is the pointcut. The piece of code  $A$  that is to be executed when condition  $C$  arises (i.e. at a join point of the pointcut) is called the *advice*.

The unit of code that defines the pointcuts and the advice related to the same concern is called the *aspect*. An aspect can also be more generally defined as a unit that encapsulates a crosscutting concern<sup>3</sup>.

Forming the final application that is executed is known as the process of *weaving*, which can be performed at runtime or compile-time, depending on the aspect language or framework implementation.

In summary, AOP aims to be as non-intrusive as possible for implementing crosscutting concerns in a system, although some code re-factoring may sometimes be necessary so as to expose the join points (depending on the language used and on its join point model). Although Perl is not natively an aspect-oriented language, the design of the authorization concern in WSRF::Lite presented in this article attempts to follow the principles of Aspect-Oriented Software Development (AOSD). It is then effectively implemented using the Perl mechanism for multiple inheritance and method dispatch. The general design guidelines could be applied to other languages and WSRF implementations, using an aspect-oriented software design and, when possible, an aspect-oriented language extension or framework.

## 4 WSRF::Lite Authorization Architecture

As described in the introduction, a design goal of this authorization mechanism is to provide a solution that does not require any modification to existing container or application code. This requirement is termed ‘non-intrusive’, meaning that the authorization code should not intrude or require any modification to the existing container or application code. The authorization concern is thought of as an aspect and, as explained below, its implementation follows patterns of AOSD. In order to illustrate how the authorization mechanism is designed and operates a simple counter, the canonical WS-Resource with WS-ResourceLifetime properties which is provided with the WSRF::Lite distribution, is described in Section 4.2 and extended to provide a role-based authorizing counter.

---

<sup>3</sup>Although it would be possible to encapsulate in the same unit sets of pointcuts and their associated advice that are not related to the same concern, this would be against the main principle of AOP, which aims to make it possible to separate concerns by encapsulating crosscutting concerns each in their own entity.

The design of an aspect for role-based authorization is presented in Section 4.1. The mechanisms used to implement it in WSRF::Lite are presented in Section 4.3.

### 4.1 Aspect for role-based authorization

The aspect presented in this paper for the role-based authorization follows the CRUD (create, read, update, delete) pattern and it comprises four pointcuts that represent create, read, update and delete operations, respectively. The CRUD pattern has been followed here as most operations on a stateful resource can be thought of as one of these type of operations and illustrates the concepts behind this work. If a more complex aspect is required, or a more fine grained level of authorization on a WS-Resource’s operations is necessary, then an aspect can be created that reflects these requirements using the techniques we describe.

Thus, in this example all operations which are ‘safe’, i.e. which do not change the state of the WS-Resource, are categorized as a ‘read’; the operations that modify the state of a WS-ResourceProperty are categorized as an ‘update’; operations that delete a WS-Resource property are a ‘deletion’; and those that add a WS-ResourceProperty are categorized as a ‘create’ operation.

The advice corresponding to each of this pointcuts consists of verifying whether the user is authorized to perform the action. If so, the execution of the join point proceeds; otherwise, an error is returned. How this verification is performed can vary according to the system and the application it is implemented as well as the policies of the resource provider or of the organization to which the user belongs.

The purpose of designing this authorization mechanism according to an aspect-oriented design is to make the implementation and evolution of these policies flexible as they can be replaced, or ‘swapped’, without affecting any other container or application code.

An example WS-Resource is now presented to illustrate how the authorization aspect is implemented in WSRF::Lite. Since Perl is not an aspect-oriented language, the mechanisms utilized to implement aspect-like behaviour are also presented.

### 4.2 Example: a Counter WS-Resource

The example `Counter` WS-Resource provided with WSRF::Lite, shown in Listing 1, is a resource that has a `count` WS-ResourceProperty. The count can be added to using the `add` operation (line 35) and the counter’s value can be subtracted from or read using the `subtract` and `getValue` operations respectively (both are not shown in the listing for brevity). In addition, the standard WSRF operations such as `GetResourceProperty` and `SetTerminationTime` (part of WS-ResourceProperty

and `WS-ResourceLifetime`) can be used to access the state of the counter `WS-Resource`.

### Listing 1. Counter WS-Resource with WS-ResourceLifetime Properties

```
1 package Counter;
2 use strict;
3 use vars qw(@ISA);
4 use WSRF::Lite;
5
6 @ISA = qw(WSRF::WSRL);
7
8 # Define our ResourceProperty count
9 $WSRF::WSRP::ResourceProperties{count} = 0;
10
11 # Prefix used when creating XML messages for count
12 $WSRF::WSRP::PropertyNamespaceMap->{count}
13     {prefix} = "mmk";
14
15 # Namespace for count property
16 $WSRF::WSRP::PropertyNamespaceMap->{count}
17     {namespace} = "http://www.sve.man.ac.uk/Counter";
18
19 # Override the default init method to set a default
20 # TT time - the init method is called when the
21 # service is created.
22 sub init {
23     my $self = shift @_;
24     alarm(60*60); # TT = 1hour
25
26     $self->SUPER::init();
27
28     $WSRF::WSRP::ResourceProperties{TerminationTime}
29     = WSRF::Time::ConvertEpochTimeToString(
30         time + 60*60 );
31     return;
32 }
33
34 # Add to the count
35 sub add {
36     my $envelope = pop @_;
37     my ($class, $val) = @_;
38
39     $WSRF::WSRP::ResourceProperties{count}
40     = $WSRF::WSRP::ResourceProperties{count} + $val;
41
42     return WSRF::Header::header($envelope),
43         $WSRF::WSRP::ResourceProperties{count};
44 }
45 ...
46
```

These operations are categorized into one of the CRUD pointcut as described in the previous section. The *read* pointcut is expressed to match the execution of `getValue` and `GetResourceProperty`, whereas the *update* pointcut matches `add`, `subtract` and `SetTerminationTime`. In addition, the *delete* pointcut matches the `Destroy` operation, part of `WS-ResourceLifetime`. Similarly, the `CounterFactory` re-

### Listing 2. The Add Method Wrapped in an Authorization Request

```
# 'Wrapped' method from original
# resource implementation
sub add {
    my $class = shift @_;
    return $class->SUPER::add(@_)
        if (isAllowedTo('update'));

    return deniedMessage();
}
```

source's `createCounterResource` operation (not presented here) is matched by the *create* pointcut. The original implementation of the counter `WS-Resource` uses the multiple inheritance mechanism of Perl via the `@ISA` variable. The following section describes how this mechanism is used for implementing the authorization aspect.

## 4.3 Implementation in Perl using multiple inheritance and method dispatch

Perl does not support AOP constructs natively<sup>4</sup>. Therefore, the aspect implementation has to be emulated using other constructs of Perl. The Perl mechanisms for supporting multiple inheritance and dynamic method dispatching help achieve the goal of non-intrusiveness in the code of the functional concerns.

Perl supports multiple inheritance through the use of the `@ISA` variable, which defines an array of classes from which a Perl class inherits. In the `Counter` class (Listing 1) this can be seen in line 6, which shows how it inherits from `WSRF::WSRL`. If more than one element is in the `@ISA` array, then Perl will follow the inheritance tree in “depth first, left-to-right order”, i.e. “Perl checks the left-most parent first, then the left most parent of that class and the left most parent of that class and so forth” [6, Ch. 6] in order to find the method requested, if it is not in the current class. When the first method matching the requested method is found it is executed. This process of searching for the right method in the inheritance tree is known as *method dispatching*.

Thus, the way the `@ISA` variable influences the method dispatch in Perl is as a simple technique for helping implement AOP concepts. Furthermore, the precedence order of the aspects is determined by the order of the declara-

<sup>4</sup>During the course of this work, the authors became aware that an Aspect module for Perl had become more stable (<http://search.cpan.org/~eilara/Aspect-0.12/>), but this module has not been used as part of this work because the major part of the work had already been done. This module might however improve further the implementation of the authorization aspect.

tions in the @ISA variable. More advanced AOP models are available in the Aspect Perl module, but this simple technique is sufficient for the purpose of adding an authorization framework without intruding on existing code.

The aspect for authorization needs to advise two sets of method calls: the custom operations (add, subtract and getValue), as presented in Section 4.3.1, and the set of standard WSRF operations, as presented in Section 4.3.2.

### 4.3.1 Weaving the aspect into the custom WS-Resource operations

The actual implementation of what was the authorization aspect at the design stage is achieved for the Counter classes methods is as follows: the Counter class is extended in a class called AuthorizingCounter that overrides each of the Counter classes methods and which ‘wraps’ them in an authorization request. If the authorization request succeeds, the call proceeds to the superclass (i.e. the Counter class). If it fails, an ‘authorization failed’ message is returned. As an example, the AuthorizingCounter’s ‘wrapped’ add method is shown in Listing 2, where isAllowedTo is a call to the authorizing mechanism, described in Section 5.

This meets the design goal of not having to modify the application code in order to support authorization and of being able to support multiple methods of authorization as this code does not prescribe how the authorization decision should be carried out.

### 4.3.2 Weaving the aspect into the standard WSRF operations

A more difficult problem is how to perform authorization on the WSRF::WSRP and WSRF::WSRL methods (such as GetResourceProperty and SetTerminationTime) without modifying any existing code. A solution could be to override the WSRF::WSRP and WSRF::WSRL methods in the AuthorizingCounter class and intercept the method calls that way, but this would lead to code duplication since this code would have to be repeated in each WSRF::WSRP and WSRF::WSRL subclass that wished to perform authorization. This code, therefore, needs to be separated out from the base class, but it cannot be placed in the WSRF::Lite code, as this would not meet the design goal of not modifying any container code. However, Perl provides a method of allowing the code to be placed in a third class, which is described in the next section.

Thus, if the WSRF::WSRP and WSRF::WSRL methods are duplicated in the classes AuthorizingWSRPResource and AuthorizingWSRLResource, respectively, and place these methods before those of WSRF::WSRP and WSRF::WSRL in

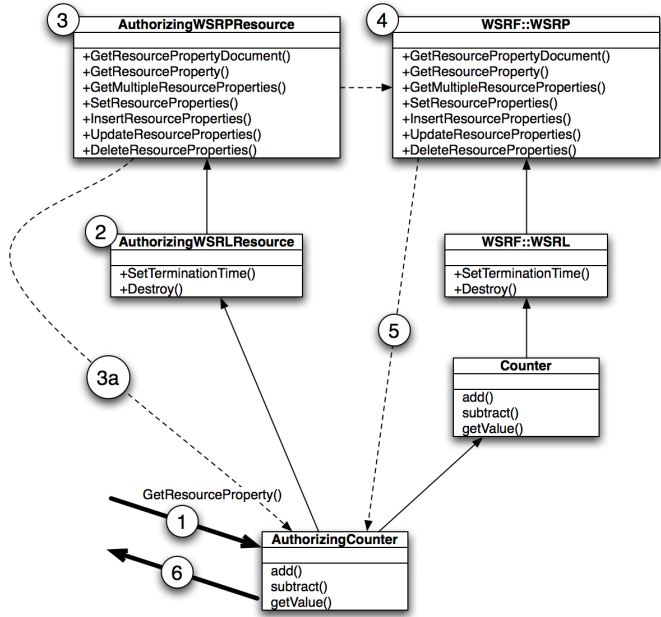


Figure 2. Authorization Inheritance and Operation

AuthorizingCounter’s inheritance tree, any calls to GetResourceProperty (for instance) will be intercepted by the authorizing method. If authorization succeeds, the request is passed to the WSRF::WSRP or WSRF::WSRL method actually requested.

Figure 2 illustrates this. In step 1, a request to carry out the GetResourceProperty operation is received by the AuthorizingCounter resource from the WSRF::Lite container. As this method is not implemented in AuthorizingCounter, Perl follows the left-hand inheritance tree, checking the AuthorizingWSRLResource (in step 2) before finding the GetResourceProperty method in the AuthorizingWSRPResource class (step 3) where an authorizing GetResourceProperty method is executed (shown in Listing 3). Note how the implementation of the authorization decision (by the isAllowedTo method) is separate, allowing the use of any authorization method.

Following a successful authorization decision, the code in Listing 3 calls the original WSRF::WSRP GetResourceProperty method (step 4) and returns the result to the AuthorizingCounter (step 5). If authorization is denied, a suitable message is returned to the requester from the AuthorizingWSRPResource (step 3a). The result of the operation, or the authorization denied message is returned to the WSRF::Lite container in step 6. This operation again meets the primary design goal of not having to modify existing container or application code in order to perform authorization at the operation level.

### Listing 3. GetResourceProperty Wrapped in an Authorization Request

```
# GetResourceProperty that intercepts
# original call and performs
# authorization request.
sub GetResourceProperty {
    my $caller = shift @_;
    my $action = 'read';
    return $caller->
        WSRF::WSRP::GetResourceProperty(pop @_)
        if ($caller->isAllowedTo($action));

    return deniedMessage($action);
}
```

### Listing 4. The AuthorizingCounter's @ISA Array

```
@ISA = qw( AuthorizingWSRLResource
            RoleBasedAuthorizer Counter );
```

#### 4.3.3 Summary

The `AuthorizingWSRLResource` and `AuthorizationWSRPResource` can be viewed, in this context, as aspects that implement the authorization concern in the system. The join points in this models are all the calls to core application functions of the resources. The pointcut and advice are expressed as the function that perform the role-based authorization; i.e. the execution of the original join point is only allowed when to authorization context has been verified.

Although the actual implementation is not, strictly-speaking, aspect-oriented, the Perl mechanisms for multiple inheritance make it possible to implement this aspect-oriented design of authorization in a non-intrusive manner with respect to the other concerns. The only intrusion in the base code is that due to the change in the `ISA` variable. The implementation of the `AuthorizingCounter`, as shown in Figure 2, can be considered as the hand-written weaving of the aspect, according to the aspect-oriented decomposition presented above. This could be enhanced by using the `Aspect Perl` module, a new version of which was only released towards the end of this project.

Therefore, this aspect-oriented approach for designing and implementing authorization in `WSRF::Lite` could be adapted and used in other `WSRF` containers, like those provided in the `Globus Toolkit` or `Apache WSRF`, for example, by using AOP extensions such as `AspectJ`, a successful AOP implementation in Java.

### Listing 5. Simple Implementation of the isAllowedTo Method

```
# Simple implementation of decision method that
# hands off authorization decision to a PDP
# over http.
sub isAllowedTo {
    my $caller = shift @_;
    my $action = shift @_;

    my $requester = $ENV{'SSL_CLIENT_DN'};
    chomp($requester);

    my $role = uri_escape(
        WSRF::WSRP::ResourceProperties($action));
    my $requester = uri_escape($requester);
    my $pdp = WSRF::WSRP::ResourceProperties('pdp');

    my $req = HTTP::Request->new(GET => '$pdp');
    $req->content_type(
        'application/x-www-form-urlencoded');
    $req->content("role=$role&requester=$requester");

    my $res = $ua->request($req);

    # Check the outcome of the response
    if ($res->is_success) {
        # Success!
        return 1;
    }

    # Denied
    return 0;
}
```

## 5 Performing Authorization Decisions

One of the main benefits of non-intrusiveness is the flexibility it entails in allowing the `WS-Resource` the ability to use different authorization mechanisms. This is necessary as the process of authorization should, in the opinion of the authors, be separated from the provision of resources. In our experience other `Grid` middleware couples these two aspects too tightly meaning the deployment and maintenance of `Grid` resources is often a laborious process. By separating the authorization process from resource provision a more dynamic and easier to manage environment can be provided. Therefore, the `WS-Resource` should not prescribe how authorization is determined, and only be interested in the result of the authorization decision, thus maintaining a clear separation of concerns in the system.

As introduced above, each method that intercepts an operation to the `WSRF::WSRP` or `WSRF::WSRL` subclass hands off the authorization decision to the `isAllowedTo` method. In order that many methods of authorization can be supported, this authorization function should not be placed in either the `AuthorizingWSRPResource`

or the `AuthorizingWSRLResource` classes (otherwise this code would have to be modified and added to for each type of authorization) or in the `AuthorizingCounter` class, as if this was the case, each resource performing authorization would have to re-implement the code, leading to duplication of code and, eventually, problems with maintenance.

## 5.1 Simple Role-Based Authorizer

The solution is to use again Perl's multiple inheritance to place the `isAllowedTo` method in the inheritance tree. To illustrate how this is done, a `RoleBasedAuthorizer` class is created that implements the `isAllowedTo` method (described in full below). This class is placed in the `AuthorizingCounter`'s `@ISA` array as shown in Listing 4, where the line should be read as 'the `AuthorizingCounter` is an `AuthorizingWSRLResource`, `RoleBasedAuthorizer` and `Counter`'. Thus, when `isAllowedTo` is called, this inheritance tree will be searched using the method dispatching process described earlier and the authorization function found and executed in the `RoleBasedAuthorizer` class.

To obtain authorization decisions, the simple role-based authorizer described here queries a policy decision point (PDP) hosted by a separate service, thus maintaining the separation of concerns in the system and allowing the authorization decision to be carried out in any way as long as the PDP implements the same interface and authorization protocol as the role-based authorizing code. The interface and protocol we have chosen to allow the querying of the PDP service uses a simple, lightweight web-style approach as, in our experience, taking a Web Service-based approach, like other service-oriented architectures use, is often too heavyweight, prone to problems with interoperation and integration and, frankly, unnecessary. Thus, in order to obtain an authorization decision the `isAllowedTo` method performs a simple HTTP query (using an HTTP GET message) of the PDP which returns either success (i.e. the entity is authorized) or failure (the requesting entity is not authorized).

In addition to implementing the `isAllowedTo` method, the `RoleBasedAuthorizer`'s `init` function initializes five WS-ResourceProperties. Four of these properties (`readRole`, `updateRole`, `deleteRole` and `destroyRole`) should be initialized values corresponding to roles contained at the PDP and the fifth (`pdp`) should be populated with the URL of the PDP itself, i.e. the address of the location carrying out authorization decisions.

The `RoleBasedAuthorizer`'s implementation of the `isAllowedTo` method is shown in Listing 5. This method operates as follows: line 8 retrieves the requesters X.509 Subject Name (SN) from the container's mutually

authenticated SSL connection with the requester<sup>5</sup>. Lines 15–17 create a new HTTP GET request to the PDP's address, and populate the message content with the parameters required by the PDP in order to make a decision. In this simple example this GET request is effectively asking 'does this role contain the requester?'. The GET is then executed through the user agent variable (created in the `RoleBasedAuthorizer`'s constructor) and the result checked. If the PDP returns a HTTP 'OK' message the authorization request has succeeded and `isAllowedTo` returns true. If something else is returned, `isAllowedTo` returns false and the authorization is denied.

Thus, as long as the PDP supports the lightweight HTTP interface described, an authorization decision may be carried out in any way the PDP seems fit and according to any policy it has implemented. For example, authorization decisions could be made by the PDP on how many times a requester has called a certain operation, or the requesters pattern of interaction with other WS-Resources using the same PDP, or through the PDP querying several sources of authority. The protocol the PDP supports can also be extended to include the details of the state of the WS-Resource so that authorization decisions can be made using a policy that provides decisions on whether a certain condition is true or if a threshold has been reached. (Note that how the PDP is implemented, operated and secured is outside the scope of this paper.)

In addition, the `RoleBasedAuthorizer` class can be customised to integrate with third-party PDPs that support XACML authorization requests or the PDP implementation provided in the Globus Toolkit, for example.

## 6 Generalization to other WSRF implementations

Although the techniques and methods presented in this paper are particular to the WSRF::Lite container and have been implemented using techniques available in the Perl programming language, the approach taken can be generalised for other WSRF containers (e.g. the Globus Toolkit's Web Service Container [9] and Apache's WSRF [2]) through the use of AOSD. For example, AspectJ, a successful AOP implementation in Java, could be used to provide the an aspect that advises method calls in the GT4.0 container to provide a simple, non-intrusive authorization mechanism. This is a piece of future work the authors intend to investigate.

---

<sup>5</sup>Although X.509 certificates have been used in this example, this method could take any authenticated identity presented to the WSRF::Lite container to make the authorization decision, allowing the solution to be used where X.509 certificates are not used.

## 7 Summary

This paper has described a method of providing role-based authorization decisions by intercepting the operations being carried out on WS-Resources hosted in the WSRF::Lite container, in a manner similar to AOP implementation techniques. This has been achieved through extending existing application and middleware code in a non-intrusive manner, i.e. without modifying any of the original WSRF::Lite container code or application code. This was necessary as this is centrally maintained and would lead to bifurcation of the code base if we were to fork it specifically for this task.

The implementation also meets the design goal of not being tied to any particular security token. Although X.509 certificates have been used in this example, the design does not prescribe that these are always used to achieve authorization. Thus, if the WSRF::Lite container were modified to accept messages via XMPP or SMTP messages, for example, the XMPP ID's or SMTP addresses could be passed to an `isAllowedTo` method that understands these authentication tokens to determine the authorization decision.

The authorization decisions themselves can be implemented in what-ever way is required for the WSRF::Lite deployment. The example presented here accesses a policy decision point (PDP) using a simple HTTP-based protocol to determine if the entity requesting a type of operation is allowed to do so. This has the advantage that the process used to make this decision is completely separated from the service providing the resource, thus maintaining a clear separation of concerns in the system.

Finally, the solution presented here also in keeping with the WSRF::Lite's philosophy of keeping the implementation simple, minimal and modular and maintaining a clear and consistent separation of concerns between the code. The result of following these principles is an extremely flexible design that allows the addition of a RBAC solution to the WSRF::Lite container.

## Acknowledgments

Michael Parkin's work is supported by a RealityGrid DTA studentship, funded by the EPSRC (GR/R67699). Bruno Harbulot acknowledges funding under PPARC Project PP/000653/1 (GridOneD).

## References

[1] R. Alfieri, R. Cecchini, V. Ciaschini, L. dell'Agnello, Á. Frohner, A. Gianoli, K. Lörentey, and F. Spataro. VOMS, an Authorization System for Virtual Organizations. In *Proceedings of the 1st European Across Grids Conference*, February 2003.

[2] Apache WSRF. Apache Web Services Project. <http://ws.apache.org/wsrf/>.

[3] B. Beckles, P. Coveney, P. Ryan, A. Abdallah, S. Pickles, J. Brooke, and M. McKeown. A User-Friendly Approach to Computational Grid Security. In S. Cox, editor, *Proceedings of the UK e-Science All Hands Meeting 2006*, September 2006.

[4] J. Brooke, P. Coveney, J. Harting, S. Jha, S. Pickles, R. Pining, and A. Porter. Computational Steering in RealityGrid. In S. Cox, editor, *Proceedings of the UK e-Science All Hands Meeting 2003*, pages 885–889, September 2003.

[5] J. Chin and P. Coveney. Towards Tractable Toolkits for the Grid: a Plea for Lightweight, Usable Middleware. Technical Report UKeS-2004-01, UK e-Science Technical Report Series, 2004.

[6] D. Conway. *Object Oriented Perl*. Manning, 2000.

[7] D. Ferraiolo, J. Barkley, and D. Kuhn. A Role-Based Access Control Model and Reference Implementation Within a Corporate Intranet. *ACM Transactions on Information and System Security*, 2(1):34–64, February 1999.

[8] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns, OOPSLA 2000*, 2000.

[9] I. Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems. In H. Jin, D. Reed, and W. Jiang, editors, *Proceedings of the IFIP International Conference on Network and Parallel Computing*, LNCS 3779, pages 2–13. Springer, November-December 2005.

[10] I. Foster, J. Frey, S. Graham, S. Tueke, K. Czajkowski, D. Ferguson, F. Leymann, M. Nally, I. Sedukhin, D. Snelling, T. Storey, W. Vambenepe, and S. Weerawarana. Modelling Stateful Resources with Web Services, v1.1, May 2004. <http://www.ibm.com/developerworks/library/ws-resource/ws-modelingresources.pdf>.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videra Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, LNCS 1241, pages 220–242. Springer, 1997.

[12] A. Nadalin, C. Kaler, R. Monzillo, and P. Hallam-Baker (Eds.). OASIS Web Services Security. Standard specification, OASIS Web Service Security (WSS) Committee, February 2006. <http://docs.oasis-open.org/wss/v1.1/>.

[13] WSRF::Lite - An Implementation of the Web Services Resource Framework. Manchester Computing, Supercomputing, Visualization & e-Science Centre. <http://www.sve.man.ac.uk/Research/AtoZ/ILCT>.