On the design, implementation and application of an authorisation architecture for web services

Sarith Indrakanti, Vijay Varadharajan* and Ritesh Agarwal

Information and Networked Systems Security Research, Department of Computing, Macquarie University, Sydney, NSW 2109, Australia
Fax: +61-2-9850-9511
E-mail: sindraka@ics.mq.edu.edu E-mail: vijay@ics.mq.edu.edu E-mail: ragarwal@ics.mq.edu.edu

*Corresponding author

Abstract: This paper proposes an authorisation architecture for web services. It describes the architectural framework, the administration and runtime aspects of our architecture and its components for secure authorisation of web services as well as the support for the management of authorisation information. The paper then describes the implementation aspects of the architecture. The architecture has been implemented and integrated within the .NET framework. The authorisation architecture for web services is demonstrated using a case study in the healthcare domain. The proposed architecture has several benefits. First and foremost, the architecture supports multiple access control models and mechanisms; it supports legacy applications exposed as web services as well as new web service-based applications built to leverage the benefits offered by the Service-Oriented Architecture; it is decentralised and distributed and provides flexible management and administration of web services and related authorisation information. The proposed architecture can be integrated into existing middleware platforms to provide enhanced security to web services deployed on those platforms.

Keywords: access control; authorisation architecture; information and computer security; web services.


Biographical notes: Sarath Indrakanti completed his Bachelor of Applied Science degree (Major in Computing) from the University of Western Sydney, Australia in 2001. He is currently a doctoral candidate in the Department of Computing at Macquarie University, Australia. His area of research is in the context of security for web services. In particular, his work focuses on providing an authorisation framework for web services. His research work is sponsored by the Australian Research Council and Microsoft Pty. Australia.

Vijay Varadharajan received his BSc (Hons) in Electronic Engineering from Sussex University, UK and PhD in Computer and Communication Security from UK in 1981 and 1984 respectively. He is the Microsoft Chairman and Professor of Computing at Macquarie University, Australia and is the Director of Information and Networked Systems Security Research. He is a member of
1 Introduction

In general, security for web services is a broad and complex area covering a range of technologies. At present, there are several efforts underway that are striving to provide security services for web services. A variety of existing technologies can contribute to this area such as TLS/SSL and IPSec. There are also related security functionalities such as XML Signature and XML Encryption and their natural extensions to integrate these security features into web service technologies such as W3C (2001) and W3C (2003).

Atkinson et al. (2002) describes enhancements to SOAP (W3C, 2003) messaging, to provide message confidentiality, integrity and authentication. Anderson et al. (2005) uses the secure messaging mechanisms of WS-Security specification to define additional primitives and extensions for the issuance, exchange and validation of security tokens within different trust domains. While there is a large amount of work on general access control and more recently on distributed authorisation (Varadharajan, 2002), research in the area of authorisation for web services is still at an early stage. There is not yet a standard specification available for web services authorisation. There are attempts by different research groups (Koshutanski and Massacci, 2002; Kraft, 2002b; Yagüe and Troya, 2002; Mont, Baldwin and Pato, 2003; Agarwal, Sprick and Wortmann, 2004; Ziebermayr and Probst, 2004) to define authorisation frameworks for the Service-Oriented Architecture (SOA). Currently most web service-based applications, which have gone through the authentication process, make authorisation decisions using application specific access control functions that results in the practice of frequently re-inventing the wheel. This motivated us to have a closer look at authorisation requirements for the SOA.

2 Authorisation requirements for the SOA

Broadly speaking, the SOA is made up of web services and business workflows composed using two or more services. These workflows are called business processes (Andrews et al., 2003). Figure 1 shows the layers comprising the SOA. In general, web services and business processes have different authorisation requirements. Authorisation services for business processes must provide orchestration services to coordinate the authorisation decisions from individual partner’s authorisation policy evaluators. Each partner must be allowed to control its own authorisation policies and also not require disclosing them to the entire workflow or to the workflow engine. Even in cases where
the binding to actual end-points of partner services happens dynamically at runtime, the authorisation architecture must be able to orchestrate the partners’ authorisation policy evaluators and arrive at an authorisation decision.

**Figure 1** Layers in the SOA

<table>
<thead>
<tr>
<th>Business Process Security Layer</th>
<th>Confidentiality, Integrity and authentication</th>
<th>Authorization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Business Process Layer</td>
<td>BPEL/AWS</td>
<td>Coordination</td>
</tr>
<tr>
<td>Web Services Security Layer</td>
<td>Confidentiality, Integrity and Authentication</td>
<td>Authorization</td>
</tr>
<tr>
<td>Web Services Technology</td>
<td>Web Services Messaging Layer</td>
<td>SOAP (extensions required to header)</td>
</tr>
<tr>
<td></td>
<td>Web Services Description Layer</td>
<td>WADL (extensions required to schema)</td>
</tr>
<tr>
<td></td>
<td>Web Services Discovery Layer</td>
<td>UDDI</td>
</tr>
</tbody>
</table>

Authorisation services for the web services layer have different design requirements as web services present a complex-layered system. For instance, a service could be a front-end to an enterprise system and the enterprise system accesses information stored in databases and files. Web services may be used by enterprises to expose the functionality of legacy applications to users in a heterogeneous environment. Or new business applications could be written to leverage benefits offered by web services.

A web service’s method may invoke one or more abstract operations, each operation having its own responsible authorisation policy evaluator. For instance, a purchase order service may have three methods – submit order, cancel order and confirm order as shown in Figure 2. Submit order and cancel order methods perform two operations – a web operation and a mail operation, and confirm order performs three operations – a web operation, a database operation and a mail operation. Each operation may have its own responsible authorisation policy evaluator to control access to the operation. For instance, the web operation may use the Java Authentication and Authorisation Service (JAAS) (Sun Microsystems, 2000) for authorising user requests, the database operation may use the Privilege Management Infrastructure (PMI) (Chadwick and Otenko, 2002) and the Mail Operation may use the Role-Based Access Control (RBAC) (Sandhu et al., 1996) mechanism for authorisation.
We envisage an authorisation framework for the SOA to provide extensions to both the security layers of web services as well as business processes as indicated by the grey coloured boxes in Figure 1. In this paper, we discuss our Web Services Authorisation Architecture (WSAA). The WSAA provides authorisation services for web services. It extends the web services security layer in the SOA. We also extend the web services description and messaging layers (indicated by dark-grey coloured boxes in Figure 1) to provide authorisation support for web services. We will describe our authorisation services for the business process layer in a separate paper.

In the next section, we outline our design principles and goals underlying the design of the WSAA. We describe some related work in Section 4. In Section 5, we describe the design of our WSAA. In Section 6, we give the extensions required to Web Services Description Language (WSDL) and SOAP to support the WSAA. In Section 7, we give the implementation details of the WSAA within the .NET framework. Section 8 demonstrates a practical application of the WSAA along with performance evaluation. We discuss the benefits of the proposed architecture in Section 9. We finally give some concluding remarks in Section 10.

### 3 Design principles for the proposed architecture

In this section, we outline some of the key design principles and goals behind our proposed architecture. We established these principles after carrying out a thorough investigation in the area of authorisation for distributed systems (Varadharajan, 2002). We also researched the available work in the web services area (Indrakanti, Varadharajan and Hitchens, 2005a; Indrakanti, Varadharajan and Hitchens, 2005b), which we discuss in Section 4.
1 Support for Multiple Access Control (AC) Models: The WSAA should be able to support a range of Access Control (AC) models. This is necessary as it is not realistic to expect every web service-based application to use the same AC model. In fact, where web services are used to expose the functionality of legacy enterprise applications, it is likely that organisations will prefer to use their currently existing AC mechanisms that they have been using before exposing them as web services. Therefore, we believe that an authorisation architecture must be generic enough to support multiple AC models including traditional Discretionary Access Control (DAC)(Abadi, 1993; Bertino, 1998; Samarati and Vimercati, 2000), lattice-based Bell-LaPadula Mandatory Access Control (MAC) (Bell and LaPadula, 1976), the Non Discretionary Access Control (NDAC)(Abrams, 1993), RBAC(Sandhu et al., 1996), Clark-Wilson model (Clark and Wilson, 1987) and Capability/Credential-based AC models(Thompson, 1999; Thompson, Essiari and Mudumbai, 2003). These AC models may be implemented using various AC mechanisms such as Access Control Lists (ACLs), capability and attribute certificate-based schemes and security label-based mechanisms.

2 Authorisation Architecture Design: Conceptually, there are two stages namely the administration phase and the runtime or the evaluation phase. The administration phase involves facilities and services for the specification of authorisation policies, updating and deleting of policies and their administration. The runtime phase is concerned with the use of these authorisation policies in the evaluation of the access requests. In general, the representation of the authorisation policy information can be different in these phases. In fact, there are at least two arguments for maintaining distinct representations of authorisation information in these two phases. Firstly, the authorisation information captured in the administration domain can be compiled out before access decision time. Secondly, it is possible to envisage different strategies for replication of the information required for administration versus the information required for access evaluation decisions at the application servers. This in turn has implications in terms of interfaces and components required in the authorisation service design. An important architectural design goal is to provide standard authorisation administration and runtime domain interfaces to different AC mechanisms. The runtime authorisation Application Programming Interface (API) is useful to invoke the authorisation components and receive an authorisation decision. The administration API is useful to group together and manage related web services and their authorisation related information.

3 Authentication: We assume that every AC mechanism supported by the WSAA provides for principal (subject) identification. This is important as it determines what privileges can be associated with the principals in the authorisation model. For instance in the case of RBAC systems, every authenticated user is allocated a role or a set of roles (where role essentially refers to a set of privileges over certain objects). So RBAC systems depend on authentication for role activation. The authentication itself may be achieved using different mechanisms. For instance, in the case of certificate-based systems, the Public Key Infrastructure (PKI) may be used to authenticate principals. In the WSAA, we assume that authentication is a prerequisite to authorisation and a principal (and its request) has undergone some reliable authentication service before being subjected to the authorisation service.
4 Authorisation Policy Evaluation: Every AC mechanism that is supported by the WSAA defines an interface or end-point (defines the input parameters as well as the output result) to what we call the Authorisation Policy Evaluator (APE). An APE is responsible for achieving end-point decisions on access control. An APE also defines a set of abstract operations such as web operations, database operations or file operations to which it provides access control.

Note: A web service method is a high-level task that the web service exposes to its clients. The WSAA provides access control indirectly to operations performed by a web service method. We map each web service method to a set of abstract operations. One or more of these operations are then mapped to an Authorisation Policy Evaluator, which is responsible for controlling access to these operations. These abstract operations help security administrators and web service developers have a common ground to map the resources (web services themselves and any other organisation resources such as databases, files, applications, etc.) to Authorisation Policy Evaluators and therefore to authorisation policies. Therefore, the developers need not have an understanding of the authorisation logic and the security administrators need not have an understanding of the application logic.

5 Authorisation Policies: Languages have long been recognised in computing as ideal vehicles for dealing with expression and structuring of complex and dynamic relationships. Over the recent years, a language-based approach to specify access control policies have (rightly) gained prominence, which is helpful not only for supporting a range of access control policies but also in separating out the policy representation from policy enforcement. Hence one of our design principles is to enable the support of a range of policy languages for specifying authorisation policies. The policy language(s) used can support both fine-grained as well as coarse-grained policies depending on the requirement. The respective authorisation policy administrators manage these policies. The design rationale is to compose the authorisation decisions from the responsible authorisation policy evaluators (involved in authorising a request) by sending the necessary credentials to them and receiving the decisions.

6 Authorisation Credentials: The WSAA provides support for defining what AC related credentials are required and how to collect them. Some AC mechanisms may pull the credentials from the respective authorities and send them to the responsible authorisation policy evaluators. For example in the Akenti architecture (Thompson et al., 1999), the policy engine component collects the relevant credentials (for the client principal) from certificate servers to make the authorisation decision. Other AC mechanisms may expect the client principal to collect the credentials from the respective authorities and push them to the responsible authorisation policy evaluators. For example in the OASIS architecture (Bacon and Moody, 2002), the client principal collects its role membership certificates from the role activation component and pushes them to the AC component to get authorised. The Praesidium Server architecture (Varadharajan, Crall and Pato, 1998) supports both push and pull models. The proposed authorisation architecture for web services supports both the push and pull models of authorisation.
4 Related work

Kraft (2002) proposed a model-based on a ‘distributed access control processor’ for web services. The main components in the authorisation model are the gatekeeper, which intercepts SOAP requests to a web service and one or more Access Control Processors (ACPs) that make the authorisation decisions for the web service. The gatekeeper itself can be an ACP. It also has the responsibility of authenticating the requesting client, combining the decisions from individual ACPs and making the final AC decision. The advantage of this model is that it supports decentralised and distributed architecture for AC. The model is generic enough to support multiple models of AC. This model however, does not provide support for administration of authorisation related information. It also does not provide support to manage Web Service Collections (WSC) and their authorisation related information using standard APIs, which our architecture provides. It is only a conceptual model and Kraft has not discussed the implementation aspects of his model.

Yague and Troya (2002) proposed a semantic approach for AC for web services. The authors defined it as Semantic Policy Language (SPL). SPL is used to create metadata for resources (Secure Resource Representation (SRR)) and generic policies without the target resource in them. A separate specification called Policy Applicability Specification (PAS) is used to associate policies to target objects at runtime dynamically when a principal makes a request. The architecture is based on the integration of a Privilege Management Infrastructure (PMI) and the SPL language features. At runtime, depending on the Source of Authorisation Descriptions (SOADs) that the Source of Authorisation (SoA) in the PMI is willing to provide to the client and the SRRs, the Policy Assistant component streamlines the SPL policies and the PAS. What is interesting in this model is that the authorisation policies can be attached dynamically based on the metadata of the resource being accessed and also be streamlined dynamically to the SOADs the SoA is willing to send, through the PMI client. The disadvantage with this model is that the authorisation policies can only be written in SPL and is based on one model of AC – the PMI, which means this model is not generic enough to support multiple AC mechanisms required by applications in a heterogeneous environment. This means legacy applications (using their own AC mechanisms) are not supported by this model. The model also does not provide management and administration support for web service objects.

Agarwal, Sprick and Wortmann (2004) proposed an AC model that combines DAML-S (Ankolekar, 2002), an ontology specification for describing web services and SPKI/SDSI, used to specify AC policies and to produce name and authorisation certificates for users. Access Control Lists (ACLs) are used to specify AC policies for web services. Each ACL has the properties keyholder, subject, authorisation, delegation and validity. Access control is defined as a pre-condition to access a web service. When querying a web service, a user sends the set of credentials required to access the web service. The user does this by using the ACL provided in the AC pre-condition of the web service provider. The user calculates the set of certificates required by making use of a chain discovery algorithm. If the client is authorised with the certificates provided, the web service returns the functional outputs sought by the client. This model is a certificate-based AC model and so is not generic enough to support multiple AC models. This means legacy applications exposed using web services cannot use different models of access control they have already been using. The ACLs in this model are simple and one cannot specify fine-grained and complex authorisation policies using this model. The
model also does not provide management and administration support for web service objects.

Ziebermayr and Probst (2004) proposed an authorisation framework for ‘simple web services’. Their framework does not consider distributed authorisation and assumes that web services provide access to data or sensitive information located on one server and not distributed over the web. The framework uses a rule-based AC model where simple rules are written for components (in which web services reside), web services and parameters of a web service method. A rule consists of a reference to a service definition, another to a user and additional rule information for parameters where necessary. When an access request comes in, the rules at these various levels are checked and an authorisation decision is made. This framework uses simple rule-based AC and so does not support multiple models of it. This means that legacy applications cannot be exposed as web services. Another disadvantage with this framework is that it cannot support authorisations for distributed web services, which have access to data and/or information over a number of web servers.

Unlike our architecture, there is no abstraction of each web service method’s function into a set of operations, in all the four authorisation models discussed in this section. This abstraction makes it easy to perform authorisation administration as discussed in Section 3.

5 Design of the web services authorisation architecture (WSAA)

Let us now briefly describe an overview of the proposed architecture (see Figure 3). The WSAA comprises of an administrative domain and a runtime domain. We manage web services in the administration domain by arranging them into collections and the collections themselves into a hierarchy. We provide administration support to manage a collection of web services. We also provide support for the arrangement (adding, removing) and movement of web services within the collections. Authorisation related components such as authorisation policy evaluators, certificate and credential authorities (provide authentication certificates and authorisation credentials) and dynamic attribute services (provide attributes required at runtime for authorisation) can be managed in the administration domain. Also security administrators can assign a set of authorisation policy evaluators to authorise requests to web services.

To make the authorisation process efficient, we have a runtime domain where the authorisation related information such as what credentials are required to invoke a particular web service and how to collect those credentials, is compiled and stored. This information is automatically compiled from time to time when necessary, using the information from the administration domain and it can be readily used by the components in the runtime domain.

The registry server located anywhere in the internet (see Figure 3) is responsible for maintaining relations between services and their service providers. When a client requests the registry server for a specific service, the latter responds with a list of web services that implement the requested service. For example a UDDI directory (Bellwood, 2004) is a registry server.
Figure 3  Web services authorisation architecture (WSAA)

5.1 System components

We define the set of Authorisation Policy Evaluators, Certificate and Credential Authorities, Dynamic Attribute Services and Authorisation Decision Composers as objects in our system. The Authorisation Manager (AZM) for an organisation is responsible to manage these components. She/he uses the Authorisation Administration API to manage them and the related data is stored in the Authorisation Administration Database (AAD). These objects are formally defined in Definitions 1 – 4.

Certificate And Credential Authority (CCA) is responsible to provide authentication certificates and/or authorisation credentials required to authenticate and/or authorise a client. For example a CCA may provide authentication certificates such as X.509 certificates (ITU-T Recommendation, 1997) or authorisation credentials such as a Role Membership Certificate (RMC) (Bacon and Moody, 2002) or a Privilege Attribute Certificate (PAC) (Chadwick and Otenko, 2002).

Dynamic Attribute Service (DAS) provides system and/or network attributes such as bandwidth usage and time of the day. A dynamic attribute may also express the properties of a subject that are not administered by security administrators. For example a nurse may only access a patient’s record if she/he is located within the hospital’s boundary. A DAS may provide the nurse’s ‘location status’ attribute at the time of access control. Dynamic attributes’ values change more frequently than traditional static authorisation credentials (also called privilege attributes) such as a Role Membership Certificate. Unlike authorisation credentials, dynamic attributes must be obtained at the time an access decision is required and their values may change within a session.

Authorisation Policy Evaluator (APE) is responsible for making authorisation decisions on one or more abstract system operations. Every APE used by an organisation may use a different AC mechanism and a different policy language to specify
authorisation policies. However, we define a standard interface for the set of input parameters an APE expects (such as subject identification, object information, and the authorisation credentials) and the output authorisation result it provides.

Authorisation Decision Composer (ADC) combines the authorisation decisions from various APE involved, using an algorithm that resolves authorisation decision conflicts and combines them into a final decision.

**Definition 1. Certificate and Credential Authority**

We define CCA as a tuple \( cca = \{ i, l, CR, pa, ra(pa) \} \), where \( i \) is a URN, \( l \) is a string over an alphabet \( \Sigma^* \) representing a network location such as a URL, \( CR \) is the set of authentication certificates and/or authorisation credentials \( cca \) provides, \( pa \) is an input parameter representing a subject, \( ra \) uses \( pa \) and gives out an output (result) that is the set of certificates/credentials for the subject.

**Definition 2. Dynamic Attribute Service**

We define DAS as a tuple \( das = \{ i, l, AT, pd, rd(pd) \} \), where \( i \) is a URN, \( l \) is a string over an alphabet \( \Sigma^* \) representing a network location such as a URL, \( AT \) is the set of attributes \( das \) provides, \( pd \) is input parameter(s) representing attribute(s) name(s), \( rd \) uses \( pd \) and gives out an output (result) that is the value of the attribute(s).

**Definition 3. Authorisation Policy Evaluator**

We define APE as a tuple \( ape = \{ i, l, pe, re(pe), OP, DAS, CCA \} \), where \( i \) is a URN, \( l \) is a string over an alphabet \( \Sigma^* \) representing a network location such as a URL, \( pe \) is the set of input parameters such as subject and object details, \( re \) is a function that uses \( pe \) and gives out an output (result) of authorisation decision. \( OP \) is the set of abstract system operations for which \( ape \) is responsible for authorisation. \( DAS \) is the set of DAS responsible for providing dynamic runtime attributes to \( ape \). \( ape \) uses these attributes to make authorisation decisions. \( CCA \) is the set of CCA that provide the credentials required by \( ape \).

**Definition 4. Authorisation Decision Composer**

We define ADC as a tuple \( adc = \{ i, l, a, pc, rc(pc) \} \), where \( i \) is a URN, \( l \) is a string over an alphabet \( \Sigma^* \) representing a network location such as a URL, \( a \) is the name of a pre-defined algorithm \( adc \) uses to combine the decisions from the individual APE. \( pc \) is an input parameter representing the decisions from the individual APE involved. \( rc \) uses \( pc \) and the algorithm \( a \) to combine the decisions and gives out an output (result) that is the value of the final authorisation decision.

The runtime domain consists of the Client Proxy, the Security Manager, the Authentication Server and the Authorisation Server (AZS) components.

**Client Proxy.** Client Proxy (CP) is an automated component that collects the required authentication certificates and authorisation credentials from the respective authorities on behalf of the client before sending a web service request and handles the session on behalf of the client with a Web service’s Security Manager. It runs on the client side.

**Security Manager.** Security Manager (SM) is a component responsible for both authentication and authorisation of the client. The SM receives the necessary authentication certificates and authorisation credentials from the CP and sends them to
the Authentication Server and the AZS for evaluation. It is responsible for managing all the interactions with a client’s CP.

**Authentication Server.** Authentication Server (ANS) receives the authentication certificates from the SM and uses some mechanism to authenticate the client. We treat ANS as a black box in our architecture as our focus is on authorisation of the client. We included this component in the web services security layer for completeness. Note that the authentication of a client is a prerequisite to its authorisation.

**Authorisation Server (AZS).** Authorisation Server (ASS) decouples the authorisation logic from application logic. It is responsible for locating all the APE involved, sending the authorisation credentials to them and receiving the authorisation decisions. Once all the decisions come back, it uses the responsible Authorisation Decision Composers to combine the authorisation decisions. Where required, AZS also collects the credentials on behalf of clients from the respective CCA.

### 5.2 Web services model

We consider a web service model where web service, web service method and web service collection are viewed as objects. WSCs are used to group together a set of possibly related web service objects. Authorisation related information can be managed in a convenient way if a set of related web service objects are grouped together in a hierarchy of collections. Hierarchical containment is widely used in current internet information systems such as web servers, file systems, URLs and is well understood. We formally define (Definitions 5–7) the web service, web service method and web service collection objects based on the model discussed in Kraft (2002).

**Definition 5.** Web service

We define a web service as a tuple $ws = \{i, b, l, \Sigma, OP_{ws}, M, MD, wsm, sm\}$, where $i$ is a non-empty string over an alphabet $\Sigma^*$ representing a globally unique identifier such as a URN, $b$ is a string over an alphabet $\Sigma^*$ representing a network protocol binding such as SOAP over HTTP, $l$ is a string over an alphabet $\Sigma^*$ representing a network location such as a URL, $\Sigma$ is a finite set of states representing the internal state of the object at a given time, $OP_{ws}$ is the set of abstract operations (for e.g. Database operation such as create table, read a row in a table, or File operation such as read and write) performed by the methods of the $ws$ object. $M$ is the set of supported web service methods; $MD$ is the metadata providing additional description for $ws$. $wsm$ is the identity of the web service manager responsible for managing the $ws$ object. $sm$ is the location of the security manager responsible for securing $ws$ object. $\Sigma, M, OP_{ws}, or MD$ can be the empty set $\emptyset$.

**Definition 6.** Web service method

We define a web service method as a tuple $m = \{i, ws, pm, rm(pm), MD\}$, where $i$ is a URN, $ws$ is the web service object the method belongs to, $OP_m$ is the set of abstract operations the method $m$ performs. $OP_m$ is a subset of the set $OP_{ws}$ defined in the $ws$ object $pm$ is the set of input parameters, string over an alphabet $\Sigma^*$, $rm$ is a function $\Sigma^* \rightarrow \Sigma^*$ that maps $pm$ onto a result string over an alphabet $\Sigma^*$ representing the output (result) or return value(s) of a computation. $pm$ and $rm(pm)$ may be the empty string $\epsilon$. $MD$ is a set of metadata providing additional description for method $m$. $OP_m$ or $MD$ can be the empty set $\emptyset$. A method $m$ has to be a member of exactly one $ws$. 
Definition 7. Web service collection

We define a WSC as a tuple $wsc = \{i, WS, WSC_{CHILDREN}, p, MD, wcm, sm\}$, where $i$ is a URN, $WS$ is a finite set of (possibly related) Web service objects in $wsc$, $WSC_{CHILDREN}$ is a finite set of Web service collections that are children of $wsc$, $p$ is the parent WSC (a WSC can have only one parent collection), $MD$ is a finite set of metadata providing additional description and semantics for $wsc$, $sm$ is the location of the security manager responsible for $wsc$. $wcm$ is the identity of the Web service Collection Manager (WCM) responsible for $wsc$. $sm$ is null for all web service collections in a hierarchy except for the root WSC, or the one without a parent $p$. In other words, if a $wsc$ object has a parent $p$, it does not have a security manager associated with it. A root WSC’s security manager is responsible for authentication and authorisation of requests to all the web services (web service objects) under its descendant collections. Figure 4 shows an example of a hierarchy of Web service collections.

Figure 4  Web services collection hierarchy

5.3 Web services administration

In this section, we discuss the administration support provided by the WSAA to manage a collection of web services. A Web Service Manager (WSM) manages web services and web service methods, and a WCM manages web service collections using the Administration API (see Figure 3). These objects are stored in the Web service Administration Database (WAD).

To effectively manage the collections, we arrange a set of related WSC objects in a tree-shaped hierarchy as shown in Figure 4. Each WSC in the hierarchy has a responsible WCM. Now the following questions arise. Who can create the hierarchy structure? Who
can add and delete Web Service (WS) objects to/from a WSC? Who has the privilege to move a WS object from one WSC to another? Before we answer these questions, we first define a root WCM (RWCM).

In a WSC hierarchy tree, the root WSC’s manager is called the Root Web service Collection Manager (RWCM). A RWCM is responsible for providing the security manager location details in the WSDL statement of every web service located under the collections she/he manages. RWCM provides these details using our extensions (discussed in Section 6.2) provided to the web service description layer.

Let us now consider an organisation with a single hierarchy (such as the one shown in Figure 4) of WSC. In Figure 4, the root WSC is WSC1 and the RWCM is WCM1. We can consider a newly initiated system to simply consist of the root WSC, WSC1 and a few WS objects under it managed by WCM1. Now what can WCM1 do with WS objects under WSC1? She/he can add new WS objects from WAD into WSC1. She/he can delete or move WS objects within the collections it is responsible for. Now there are two basic issues to consider.

1. Who decides the location of a WS object (and how is the location changed)?
2. Who decides the shape of the tree itself?

5.3.1 Web service object location

Consider the example shown in Figure 4. A WS object WS1 resides in WSC3 managed by WCM3. Who has the authority to move a WS object from one WSC to another? For example who has the authority to move WS1 from WSC3 to WSC5? We can immediately rule out WCM4 and WCM5. WCM5 has no authority over WS1 at all (even if the point of the exercise is to give it the authority). A WCM should not be able to arbitrarily assume authority over another WSC’s objects. Can WCM4 make the transfer? WCM1’s ability to do the transfer is a viable option. WCM1 and WCM2 may also be able to make the change. WS1 resides in their descendant collection WSC1. So it could be argued that they should be able to carry out the operation. In fact, it could be argued that only they could do it, as once a WCM is given responsibility for a WS object, they should not divest themselves of it. Hence, WCM1 should not be able to carry out the operation. Alternately, it may be desired to restrict the authority to make all such changes to only a single entity, in this case, the RWCM – WCM1. So we have three possible options, each reflecting different organisational approaches:

1. The WCMs of the ancestor WSCs of the WSC in which the WS object being moved resides,
2. Both the WCM of the immediate WSC in which the WS object resides and the WCMs of the ancestor WSCs of the WSC in which the WS object resides,
3. The RWCM.

It can be argued that a fourth option exists, in which only the WCM of the immediate WSC (in this case WCM1) can move the object. But as the meaning of the hierarchy is that a WS object resides in both its immediate WSC and in the ancestor WSC, this seems counter-intuitive. In the example, if WCM3 is to have the authority, then it should not be placed as a subordinate to the other WCMs.
Option 1 presents a problem in the case of WS objects under a root WSC, as there are no ancestors in this case. This can be solved either by any WS objects to actually reside in the root WSC or by making an exception for the root WSC (effectively combining options 1 and 3). In some sense the RWCM is a super user. Some organisations may not wish to have such a user, but for our purposes, this could be handled by having multiple WSC hierarchies in an organisation each managed by different RWCM.

The next question is, are there any limits to the destination of the WS object while making a move within the hierarchy? The most restrictive is to allow a WCM to move a WS object to a WSC under his/her control (the immediate WSC managed and other descendant WSCs). Less restrictive options are to allow a WS object to be passed to the parent WSC or any other ancestral WSCs. The target WSC to which a WS object is moved in the tree may be:

1. any WSC in the tree
2. the WSC of the WCM making the move or any descendant WSC
3. as in 2 plus the immediate parent WSC
4. as in 2 plus any ancestor WSCs.

Combining this with the example above, we would have the rule that a WCM can relocate any WS object that resides in their immediate WSC or any descendant WSCs to a target WSC that is either the WSC managed, any descendant WSC or any ancestor WSC. Note that this also gives us a rule placing a new WS object into a WSC. Let us again consider our example from Figure 4 of moving WS1 object from WSC3 to WSC5. Under rule 3 above, only WCM1 or WCM2 could perform the operation. Only under option 2 on which WCM can make the move and option 1 on destination could WCM3 move the WS1 object. Even with option 2 on which can make the move, any other choice of options but option 1 for destination control prevents WCM3 from making the move. Another possible option might be that WS objects could be transferred to sibling WSCs. This would then allow WCM2 to move WS1 object from WSC3 to WSC5. We find it likely that some organisations would not allow such transfers of responsibility without the involvement of more senior authority. However, it would allow WSC management structures to be created where WS objects could be transferred between the WSCs of a specified group of WCMs without allowing unrestricted transfers to any point in the hierarchy tree. This could be achieved with the combination of any of the options 2–4 mentioned above.

5.3.2 Shape of the tree

Let us now briefly consider the creation and deletion of WSCs and the appointment of WCMs for the WSCs. Initially, a root WSC with a RWCM is created. RWCM appoints himself/herself to the root WSC. Organisations could also use a simple rule where manipulating the shape of the tree be restricted to the RWCM. A similar rule can apply to delete WSCs from a tree. In this case, the RWCM appoints the WCMs for all the WSCs in the tree.

In a less restrictive scenario, any WCM in the tree should only be able to create and delete the descendant WSCs, as well as appoint a WCM for the WSC she/he manages.
Manipulating the WSCs for which the WCM has no responsibility is obviously undesirable.

5.4 Authorisation administration and policy evaluation

A WSM is responsible to manage the authorisation related information for the web services she/he is responsible for. We consider a web service method to be a high-level task that is exposed to clients. Each task (method) is made up of a number of system operations. These operations can be of different abstract types. For instance, each method of the purchase order service shown in Figure 2 performs one or more of these three operations – web operation, database operation and mail operation. Each of these operations has a responsible APE. Web operations are authorised by using the Java Authentication and Authorisation Service; database operations are authorised by the Privilege Management Infrastructure (PMI); and the mail operations are authorised by using a RBAC mechanism. It is reasonable to assume that a WSM knows the set of organisation-defined task (such as web operation in the case of purchase order service) a web service under his/her control performs. Similarly a WSM knows the set of operations each of these tasks (methods) perform. The WSM associates these tasks to the web services and their methods. Then using the APE definitions from Authorisation Administration Database (AAD), the administration domain automatically associates APEs to web service methods. The association to APEs happens, based on the tasks WSM associates with web service methods. For instance, if APE1 provides authorisation for web operations and APE2 provides authorisation for mail operations and WSM chooses web operation and mail operation as task for ‘submit order’ method of purchase order service. In this case, APE1 and APE2 are automatically associated with the ‘submit order’ method of purchase order service. This association is made in the Web Service Method Authorisation (WSMA) object. The object is stored in AAD (see Figure 3). Therefore, we are able to separate the authorisation related responsibilities from WSMs who are typically involved in developing web services.

Similar to web service methods, a web service can also have one or more APEs responsible for web service level authorisation. A WSM may associate one or more APEs to a web service manually should she/he decide to do so. This association is made in the Web Service Authorisation (WSA) object. The object is stored in AAD (see Figure 3). We give this provision to enable WSMs (or web service developers) to associate coarse-grained authorisation with the web services they manage. For instance, APE3 can be associated to purchase order service. Web service level policies are first evaluated before its method level authorisation policies are evaluated. A web service’s APEs evaluate web service level authorisation policies. These policies typically be relatively coarse-grained and not be as fine-grained as method level authorisation policies. A WSM may choose to create a new ADC for one or more web services she/he manages or may decide to use one from the set of existing ADCs from AAD if it serves the purpose.

Similar to web services and their methods, a WSC can also have one or more APEs responsible for authorising access to the collection itself. A WCM may associate one or more APEs to a WSC and manually she/he should decide to do so. This association is made in the Web Service Collection Authorisation (WSCA) object. The object is stored in AAD (see Figure 3). We give this provision to enable WCMs to associate coarse-grained authorisations with the WSCs they manage. Collection level policies are first evaluated before web service level authorisation policies are evaluated. This is
because these policies will typically be coarse-grained (and therefore typically takes less time to evaluate) when compared to relatively finer-grained web service and web service method level authorisation policies. A WSC’s APEs evaluate collection level authorisation policies. Every root WSC has an ADC associated with it, which is responsible for combining the decisions from all the APEs involved. The coarse-grained authorisation policies for all the relevant ancestor WSCs (of an invoked web service) are first evaluated, followed by the web service level authorisation policies, and finally the fine-grained web service method level policies are evaluated. The coarse-grained policies are first evaluated before the finer-grained policies as it helps reduce the computing cost. If the client is not authorised by a coarse-grained policy, access can be denied straight away. For example with reference to Figure 4, when a client invokes WS1’s method M1, WSC1’s authorisation policies are first evaluated by APE1 and APE2, followed by WSC2 (APE3) and then WSC3 (APE4) policies. If APE1, APE2, APE3 and APE4 give out a positive decision, WS1’s authorisation policies are evaluated by APE6. If APE8 gives out a positive decision, then finally M1’s authorisation policies are evaluated by APE7 and APE8. WS1’s ADC, ADCWS1 combines the decisions from APE6, APE7, and APE8 and if the final decision is positive, WSC1’s ADC, ADCWSC1 combines the decisions from APE1, APE2, APE3, APE4 and ADCWS. If the final decision from ADCWSC1 is positive, the client will be authorised to invoke WS1’s method M1.

WSMs manage WSA and WSMA using the Authorisation Administration API. Similarly WCMs manage WSCA objects using the Authorisation Administration API. WSA, WSMA and WSCA objects are stored in AAD (see Figure 3). We define WSA, WSMA and WSCA objects in Definitions 8–10.

**Definition 8. Web Service Method Authorisation**

We define WSMA as a tuple \( wsma = \{i, m, APE_m\} \), where \( i \) is a URN, \( m \) is the URN of the method for which \( wsma \) object is defined. \( APE_m \) is the set of URNs of the APE responsible for authorising requests from a client to the method \( m \).

**Definition 9. Web Service Authorisation**

We define WSA as a tuple \( wsa = \{i, ws, APE_{ws}, adc_{ws}\} \), where \( i \) is a URN, \( ws \) is the URN of the Web Service for which \( wsa \) is defined. \( APE_{ws} \) is the set of URNs of the APE responsible for authorising requests from a client to \( ws \). \( adc_{ws} \) is the URN of the ADC for \( ws \). It is responsible to combine the decisions from APE in the set \( APE_{ws} \).

**Definition 10. Web Service Collection Authorisation**

We define WSCA as a tuple \( wasca = \{i, wsc, APE_{wsc}, adc_root\} \), where \( i \) is a URN, \( wsc \) is the URN of the WSC for which \( wasca \) object is defined. \( APE_{wsc} \) is the set of URNs of the APE responsible for \( wsc \). \( adc_{root} \) is the URN of the ADC for \( wsc \). If \( wsc \) is not a root WSC, then \( adc_{root} \) is null. In other words, \( adc_{root} \) exists only for a root WSC.

**5.5 Runtime authorisation data**

We have now addressed who assigns (and how) APE and ADC to web services and WSCs. The next question is, at runtime, how does a client know (where necessary) where to obtain from, the required authorisation credentials and dynamic runtime attributes before invoking a web service? What are the responsible APE (and the credentials and attributes they require), CCA (and the credentials they provide) and the DAS (and the
attributes they provide)? How does the AZS know what the set of responsible ADC (adc_{ws} and adc_{root}) for a particular client request is?

To answer these questions, we have an Authorisation Runtime Database (ARD) in the runtime domain. ARD consists of the runtime authorisation related information required by the clients (CP) and the AZS. Credential Manager (CRM) is an automated component that creates and stores the authorisation runtime information in ARD using the information from WAD and AAD databases. The runtime authorisation information consists of three tuples defined in Definitions 11–13. CRM is invoked from time to time, when a web service object is added or deleted to a collection, moved within a hierarchy of collections or when the shape of the tree itself changes, to update these tuples in ARD.

**Definition 11. Method-Credential-CCA tuple**

We define the Method-Credential-CCA tuple as $mcc = \{i, m, CR, cca, ape\}$, where $i$ is a URN, $m$ is a web service method to which the tuple is defined, $CR$ is the set of authorisation credentials to be obtained from the $cca$ to get authorised to invoke $m$. This means each $m$ object can have one or more of these (tuple) entries in ARD. $ape$ is the APE that requires these credentials.

**Definition 12. Method-Attribute-DAS tuple**

We define Method-Attribute-DAS tuple as $matd = \{i, m, AT, das, ape\}$, where $i$ is a URN, $m$ is a web service method to which the tuple is defined, $AT$ is the set of attributes to be obtained from the DAS $das$. This means each $m$ object can have one or more of these (tuple) entries in ARD. $ape$ is the APE that requires these attributes.

**Definition 13. WS-ADC tuple**

We define WS-ADC tuple as $wsd = \{i, ws, adc_{ws}, adc_{root}\}$, where $i$ is a URN, $ws$ is a web service, $adc_{ws}$ is the ADC for $ws$. $adc_{root}$ is the ADC for the root WSC in which $ws$ resides.

**5.5.1 CRM algorithm**

CRM creates these tuple entries in the ARD using the following algorithm:

1. For every web service method defined in the Web services Administration Database (WAD), it generates a list, APE\_List of responsible APE using its wsma tuple and related wsa and wsca tuples from AAD,
2. For each APE, ape in APE\_List,
   2.1 for each $cca$ responsible for the ape, an $mcc$ tuple is created in the ARD
   2.2 for each $das$ the $ape$ uses, an $matd$ tuple is created in the ARD.
3. For every web service object, CRM creates a $wsd$ tuple entry in the ARD using the $wsa$, $wsc$ and $wsca$ tuples of the root WSC in which the $ws$ resides.

When a web service object is placed and/or moved within a WSC in a tree, the set of APE responsible for authorising a client’s request changes. Similarly, the set of CCA and DAS responsible also changes. For example with reference to Figure 4, when WS$_1$ moves from WSC$_3$ to WSC$_5$, the set of responsible APE for WS$_1$’s method M$_2$ changes from \{APE$_1$, APE$_2$, APE$_3$, APE$_4$, APE$_6$, APE$_7$, APE$_9$\} to \{APE$_1$, APE$_2$, APE$_3$, APE$_5$, APE$_6$, APE$_9$\}.
APE$_{1}$, APE$_{3}$}. Once the move is made, CRM is automatically invoked and it updates the ARD with the necessary $mcc$ and $matd$ tuple entries for each method of WS$_{1}$. In this example, the $wsd$ tuple remains the same before and after the update. The responsible ADC before and after the move will still be ADC$_{WSC1}$ and ADC$_{WS1}$.

5.6 Authorisation algorithms

The WSAA supports three authorisation algorithms. The first, push-model algorithm supports authorisations where a client’s CP, using the information in WS-AuthorisationPolicy (defined in Section 6.1), collects and sends the required authorisation credentials (from CCA) and attributes (from DAS) to a web service’s security manager. The second, pull-model algorithm supports authorisations where the AZS itself collects the required authorisation credentials from CCA and the APE themselves collect the required attributes from DAS. The AZS in this case uses the runtime objects (tuples) information from the ARD to be able to do so. The third, combination-model supports both the push and pull models of collecting the required authorisation credentials and attributes.

An organisation must deploy one of these algorithms depending on the AC mechanisms used. If all the AC mechanisms used by the set of APEs are based on a pull-model, then the organisation must deploy the pull-model algorithm. If all the AC mechanisms used are based on a push-model, then the organisation must deploy the push-model algorithm. However, when some of an organisation’s APEs use the pull-model and others use the push-model, the combination-model algorithm must be deployed.

When the combination-model algorithm is deployed by an organisation, the organisation’s Authorisation Manager (AZM) may arbitrarily decide whether the authorisation credentials required from a CCA and dynamic attributes required from a DAS for each web service method, web service as well as WSC level APEs, are fetched by a CP (push-model) or by the authorisation components themselves (pull-model). The AZM may decide to give the entire responsibility of fetching the required credentials and attributes to the CP or to authorisation components or share responsibility of fetching credentials and attributes amongst the CP and the authorisation components. This information is reflected in a web service’s WS-AuthorisationPolicy, defined in Section 6.1.

5.7 Sequence of steps involved in authorisation

We show the sequence of steps involved in authorising clients to web services, using a system sequence diagram (Figure 5). Steps involved in the push and the pull models of authorisation are not shown here, as they are a subset of the combination-model authorisation algorithm. In the push-model, we do not have steps 13.1, 14, 15, 17 and 18 and in the pull-model we do not have steps 6 and 7. However in the pull-model, we still have steps 4 and 5 to fetch the authentication certificates, where required.
Figure 5  Sequence of steps involved in combination model authorisation
A client using an interface queries a registry server such as a UDDI directory for a web service.

The registry server looks up for this web service and returns a list of appropriate web service(s).

The client chooses a web service WS₁ and invokes a method on WS₁.

The CP intercepts the web service request and retrieves WS₁’s WSDL statement. It locates WS₁’s WS-SecurityPolicy (for authentication) and WS-AuthorisationPolicy (for authorisation) statements. Using the information in WS-SecurityPolicy and WS-AuthorisationPolicy, the CP requests the required authentication certificates and/or authorisation credentials from a CCA.

The CCA sends the required certificates or credentials to the CP. Steps 4 and 5 are repeated until all the required authentication certificates and/or authorisation credentials are collected.

Using the information in WS-AuthorisationPolicy, the CP requests the required runtime attributes from a DAS.

The DAS sends the required attributes to CP. Steps 6 and 7 are repeated until all the required attributes are collected.

The CP sends the web service (method) request along with the collected authentication certificates, authorisation credentials and runtime attributes to WS₁’s security manager. It finds the security manager’s location using WS₁’s WSDL statement.

Then the security manager sends the authentication certificates to the authentication server.

The authentication server gets back with a decision. If authentication fails, then security manager sends an ‘authentication fail’ message to the CP and the access is denied to the client (this step is not shown in Figure 5). If authentication is successful, the algorithm continues as follows:

The security manager sends the authorisation credentials and attributes received from the CP to the AZS using the authorisation API.

The AZS sends a request to the database broker component for the sets of tuples (runtime objects) for the requested web service, WS₁. The database broker fetches these details from the ARD.

AZS receives the sets of requested tuples. Using them, the AZS creates a list of responsible APE, in the APEₐₙₜₑₐₗₜᵦₜ, for the method requested.

For each APE in APEₐₙₜₑₐₗₜᵦₜ, the AZS creates a list of responsible CCA, in the CCAₐₙₜₑₐₗₜᵦₜ.

From each CCA in CCAₐₚₑₑₐₗₜᵦₜ, the AZS requests the set CR of necessary credentials on behalf of the client.
The CCA sends the set CR of credentials to the AZS. Steps 14 and 15 are repeated until all the credentials are collected. These steps are only executed if the CP did not already send the credentials required by the APE.

The AZS, using the information in the tuples, sends the appropriate credentials to an APE in the APEList, along with the subject (client authentication details) and target (resource) details.

The APE, if necessary, requests a DAS to collect the set AT of runtime attributes required by its authorisation policies.

The DAS sends the set AT of attributes to the APE. Steps 17 and 18 are repeated to collect the required runtime attributes from all the DAS involved. They are executed if the CP did not already collect and send the required attributes.

The APE evaluates its authorisation policies using the credentials and/or attributes and then sends its authorisation decision to the AZS. Steps 13.1 to 19 are repeated for all the APE in APEList.

The AZS locates the root web service collection’s ADC, $adc_{root}$ using the necessary tuple (received from the database broker) and sends the set of authorisation decisions from all the APE involved to it.

The ADC, $adc_{root}$ combines the decisions using a pre-defined algorithm and sends the final authorisation decision to the AZS. As mentioned earlier in Section 5.4, $adc_{root}$ first delegates the task of combining the web service and web service method level authorisation decisions to WS1’s ADC, $adc_{WS1}$ and then combines that decision with WSC level authorisation decisions.

The AZS gets back to the security manager with the final authorisation decision. If authorisation fails, then the security manager sends an ‘authorisation fail’ message to the CP and the access is denied to the client (this step is not shown in Figure 5). If authorisation is successful, the algorithm continues as follows:

The security manager acts as a broker for the client’s request and sends the request to the appropriate web service (WS$_i$).

WS$_i$ gets back with a result to the security manager.

The security manager sends the result back to the CP.

The client (interface) receives the final result from the web service via the CP.

6 Extensions to the web service description and messaging layers

We require extensions to the web service description and messaging layers of the SOA to support the WSAA. We provide extensions to the SOAP header (messaging layer) to carry authorisation related credentials and attributes. We extend WSDL (description layer) to include a WS’s-AuthorisationPolicy as well as the location of its security manager.
6.1 WS-authorisationPolicy statement

WS-SecurityPolicy (Della-Libera et al., 2002) statement consists of a group of security policy ‘assertions’ that represent a web service’s security preference, requirement, capability or other property. Similarly, we define WS-AuthorisationPolicy as a statement that contains a list of authorisation assertions. The assertions include what credentials (and from which CCA) and attributes (and from which DAS) a client’s CP has to collect before invoking a web service. WS-PolicyAttachment standard (Bajaj et al, 2004) can be used to link the WS-AuthorisationPolicy to a web service’s WSDL statement. Figure 6 shows the XML schema skeleton for WS-AuthorisationPolicy.

Figure 6  WS-AuthorisationPolicy XML schema skeleton

```
<WS-AuthorizationPolicy>
    <URN>WS1</URN>
    <URL>WS1's URL</URL>
    <WSDL-URL>WS1's WSDL location</WSDL-URL>
    <WS-Method>WS1's methods</WS-Method>
            <Credentials>
                <CCA>CCA's URL</CCA>
                    <Credential>credentials provided by CCA</Credential>
                        <Attributes>
                            <DAS>DAS's URL</DAS>
                                <Attribute>DAS's attributes provided</Attribute>
                                    </Attributes>
                            </Attributes>
                        </Attributes>
                    </Credential>
                </Credentials>
            </Attributes>
        </WS-Method>
    </WS-AuthorizationPolicy>
```

6.2 Security manager location

When a client wants to invoke a web service WS1, the CP component requires WS1’s security manager’s location. Therefore, we need to give this information in WS1’s WSDL statement. We introduce a new element called SecurityManager to the WSDL document. XML schema skeleton for security manager element and an example WSDL statement are shown in Figure 7.

6.3 SOAP header extension

WS-Security (Atkinson et al., 2002) enhancements for confidentiality, integrity and authentication of messages have extended SOAP header (SOAP-SEC element) to carry related information. Similarly we suggest extending SOAP header to carry authorisation credentials and attributes to carry authorisation related information. When a client wants to invoke a web service object, the CP creates an authorisation header object and adds it to SOAP header before making a SOAP request. We show the XML schema skeleton for extended SOAP header in Figure 8. The SOAP Authorisation (SOAP-AUTHZ) header (shown in bold) consists of the list of credentials and attributes the CP collects on behalf of the client.
Figure 7  Extended WSDL schema skeleton example

```xml
<definitions name="StockQuote">
  ...
  <securityManager>
    <location>http://example.com</location>
  </securityManager>
  <types>
    ...
  </types>
  <message name="GetLastTradePriceInput">
    ...
  </message>
  <portType name="StockQuotePortType">
    ...
  </portType>
  <binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
    ...
  </binding>
  <service name="StockQuoteService">
    ...
  </service>
</definitions>
```

An example WSDL document with the extension.

Figure 8  XML schema skeleton for SOAP authorisation header

```xml
<SOAP-ENV>
  <SOAP-ENV:Header>
    <SOAP-SEC/>
    <SOAP-AUTHZ/>
    < Credentials>
      <CCA/>
      <CCA-URN/>
      <Credential/>
    </Credentials>
    <Attributes>
      <DAS/>
    </Attributes>
  </SOAP-AUTHZ>
  <SOAP-ENV:Header>
  <SOAP-BODY>
  ... 
  </SOAP-BODY>
</SOAP-ENV>
```
7 Implementation of the WSAA

We have implemented the WSAA as a middleware layer within the .NET framework (Microsoft Corporation, 2005). We outline the high-level four-tier architecture overview of the implementation in Figure 9.

Figure 9  Four-tier architecture overview

Presentation tier: We provide web interfaces to all the managers (AZM, WCM and WSM). When a manager is authenticated to the administration domain, she/he is redirected to the appropriate web page. We built the (ASP.NET) web pages using the Visual Studio .NET (2003 version) Software Development Environment (SDE).

Service tier: We have three web services that provide the interfaces to the authorisation logic. They are AZM-Service, WCM-Service and WSM-Service. They expose the WSAA administration functionality required for the AZM, WCM and WSM managers. The (ASP.NET) web services are implemented using the Visual Studio .NET (2003 version) SDE.

Business logic tier: The authorisation architecture logic is implemented as .NET components using the C# programming language. The .NET components are once again built using the Visual Studio .NET (2003 version) SDE. The required administration and authorisation API as well as the extensions required to WSDL and SOAP are implemented in this tier. The authorisation logic is neatly separated into two separate namespaces. The namespaces and C# classes are shown below.


Classes:

AuthorisationManager.cs
Classes:
ClientProxy.cs
SecurityManager.cs
AuthenticationServer.cs
AuthorisationServer.cs
CertificateAndCredentialAuthority.cs
DynamicAttributeService.cs
AuthorisationPolicyEvaluator.cs
AuthorisationDecisionComposer.cs

Data tier: We defined normalised SQL tables for both the administration and runtime database objects (for the logically separate WAD, AAD and ARD databases). We use one single Microsoft SQL 2000 Server to store all the three logically separate databases.

8 Application of the WSAA

We made use of a service-oriented healthcare application to practically demonstrate the features of the WSAA. In this section, we briefly discuss a four-tier architecture (refer to Figure 9) of the healthcare application.

Presentation tier: The healthcare application is exposed to clients such as doctors, patients and nurses using ASP.NET web pages built using the Visual Studio .NET (2003 version) SDE. The security manager is placed in the firewall zone and parses the incoming SOAP requests on HTTP. It relies on the AZS to authorise the client requests. The AZS uses the authorisation interface to contact the necessary runtime authorisation components. Once the final authorisation decision is made, the AZS gets back to the security manager with that decision.

Service tier: The application’s logic is implemented as ASP.NET web services. EssentialDetailsWS, ClinicDetailsWS and HealthDetailsWS are the web services used to read, write, create or delete the essential, clinic and health details of a patient in his/her record. Another web service, LoginWS, is used to authenticate application users. Users authenticate themselves using their user names and passwords (using a web interface).

Business logic tier: The healthcare application logic is implemented in this layer as .NET components (C# classes). We defined an XML schema for the patient records. We implemented the logic to read, write and modify XML patient records in this layer.
**Data tier:** Once again, we made use of the Microsoft SQL 2000 Server to store the patient medical records. The medical records in the system are stored in XML format using the SQLXML technology (Microsoft Corporation, 2004) in the SQL 2000 Server.

### 8.1 Management of web services

Figure 10 shows the management hierarchy of WSCs in a medical practice. We used the access control requirements for a medical practice given by the specification in (Sim, 2002) to implement the healthcare application. As shown in the figure, we have the web services arranged into a hierarchy. We have two collections of web services – Security and Healthcare Application collections, both in the root collection of medical practice. The security collection has only one web service – LoginWS, which is used to authenticate application users. The Healthcare Application is made up of three web services – EssentialDetailsWS, ClinicDetailsWS and HealthDetailsWS, which are used to read, write, create or delete the essential, clinic and health details of a patient from his/her record.

**Figure 10** Healthcare application web services management hierarchy
The root WSC medical practice is associated with the APE. APE\(_1\) relies on the CCA\(_1\) for authorisation credentials and the DAS\(_1\) for dynamic runtime attributes. CCA\(_1\) provides a signed IP address of the client’s workstation as a credential and DAS\(_1\) provides the location (inside or outside the hospital) of the client. APE\(_1\) uses the credential and attribute provided to authorise clients. If the signed IP address provided is within the list allowed and the location is ‘inside’ the hospital, APE\(_1\) gives out positive authorisation to access any WSC or web service under the root WSC medical practice.

The web service collection, security, makes use of APE\(_2\) for authorisation. APE\(_2\) relies on CCA\(_2\) for providing authorisation credentials. CCA\(_2\) provides a Role Membership Certificate (RMC) for clients. APE\(_2\) gives positive authorisation to the roles patient, receptionist, pathology collector, practice manager, nurse, doctor and medical director defined for the healthcare application. If the RMC provided by CCA\(_2\) contains either of these roles, then positive authorisation is provided to the web service (LoginWS) under the ‘Security’ WSC. It is noted that LoginWS and its method Login() do not have any APE associated with them.

Once again, the WSC, healthcare application, makes use of APE\(_2\) for authorisation. APE\(_2\) only gives positive authorisation to the roles patient, receptionist, pathology collector, nurse, doctor, medical director and practice manager.

APE\(_3\) authorises user access to the EssentialDetailsWS web service. APE\(_3\) makes use of CCA\(_2\) and DAS\(_2\) for providing credentials and dynamic runtime attributes required for authorisation. DAS\(_2\) is a simple service that provides the time at which the client request is made. APE\(_3\) in this case only gives positive authorisation to the roles receptionist, doctor and medical director to access the web service using RMC provided by CCA\(_2\). The added constraint is that access is only allowed between 9am and 5pm. APE\(_3\) also authorises user access to the ClinicDetailsWS web service. APE\(_3\) in this case only gives positive authorisation to the roles pathology collector, nurse, doctor, medical director and practice manager. Finally, APE\(_3\) also authorises user access to the HealthDetailsWS web service. APE\(_3\) gives positive authorisation only to the doctor and medical director roles to the HealthDetailsWS web service.

APE\(_4\) is responsible to authorise requests to the methods of all the three web services – EssentialDetailsWS, ClinicDetailsWS and HealthDetailsWS. APE\(_2\) relies on CCA\(_3\) for providing authorisation credentials. In particular, CCA\(_3\) looks up the users file and returns a string (user’s role). APE\(_4\) uses the credential (role) and provides fine-grained access control to patient records, which are stored in XML. APE\(_4\) provides an interface to the XML Access Control Language (XACL) (Kudo and Hada, 2000) policy evaluation engine. The engine provides fine-grained access control to a subject who is a doctor for instance (role) to a patient’s XML medical record (target) after evaluating the policies which are written in XML.

When LoginWS is requested, ADC\(_1\) combines the authorisation results from APE\(_1\) and APE\(_2\) to give out the final authorisation to the client using an algorithm. The algorithm is simple. It gives ‘true’ as final authorisation if both APE\(_1\) and APE\(_2\) give out positive authorisation.
Design, implementation and application of an authorisation architecture

EssentialDetailsWS, ClinicDetailsWS and HealthDetailsWS make use of ADC₂. When a method in either of these web services is invoked, ADC₂ combines the authorisations from APE₃ and APE₄. Once again the authorisation algorithm is simple. It gives ‘true’ as final web service level authorisation if both APE₃ and APE₄ give out a positive authorisation. Then root WSC’s medical practice ADC₁ uses its algorithm to combine the authorisations from APE₁, APE₂ (for Healthcare Application WSC) and ADC₂. If the final authorisation given by ADC₁ is ‘true’, then the client access is authorised on the requested method.

8.1.1 Administration Screen Shots

When a security administrator (either AZM, WCM or WSM) logs into the administration domain, a list of roles that she/he is a member of, appears in a list box as shown in Figure 11. The administrator chooses an appropriate role and logs into the administration domain. Depending on the role chosen, a web page, with the functionality provided to the role, is displayed to the security administrator. For instance, as seen in Figure 11, John could choose from the roles AZM1, WCM1 or WSM1. He is assigned to all the three administration roles.

When a security administrator logs in as a WCM and chooses to create new WSC, the web page shown in Figure 12 appears. The page shows the hierarchy of web services and WSC used by an organisation (in this case a hospital). The WCM can create new WSC by clicking on the ‘Add WSC’ button. The WCM can also add a new web service into one of the collections she/he is responsible for. S/he can do this by selecting a WSC and clicking on the ‘Add Web Service’ button. The web service is added into that selected WSC. She/he can also create a root WSC hierarchy by clicking on the ‘Create Hierarchy’ button. She/he can then add new WSCs and web services under that root WSC. Finally, if the logged-in WCM is a RWCM, she/he has the privilege to create the security manager information in the WSDL document of all the web services under his/her root WSC. She/he can do this by selecting a web service and clicking on the ‘Add SM Information in WSDL’ button.

If the WCM chooses to update an existing collection, the web page shown in Figure 13 appears. The WCM can select any WSC in the hierarchy and update its information by clicking the ‘Update WSC’ button. A WCM can delete any WSC that she/he is responsible for, by selecting the WSC and clicking on the ‘Delete WSC’ button. Note that when a WSC is deleted, all the web services contained within it are also removed from the WSC. However, the web services (tuples) definitions are still available in the ARD for use in other WSC. A WCM can also delete any (one) web service within a WSC she/he is responsible for, by selecting the WS and clicking on ‘Delete WS’ button.

A WCM can also move a web service from one WSC to another if she/he has control over both the WSCs. She/he can do this as follows:

1. select a web service and then click the ‘Move WS’ button
2. select the destination WSC by clicking on it in the WSC hierarchy tree
3. click on the ‘Select Destination WSC’ button.
Figure 11  Administration domain login web page
Figure 12 WSC management tree (create)
Figure 13  WSC management tree (Update)
8.1.2 Runtime screen shots

Figure 14 shows the sequence of authorisation steps involved in authorising a user to access the Login() method of the LoginWS web service. The LoginWS uses the push-model of authorisation. Therefore, the CP collects the credentials and attributes required for authorisation. CCA1 and CCA2, are first invoked and then the DAS1, is invoked. The security manager authenticates the client using the authentication server and then authorises the client using the AZS. In this case, both authentication and authorisation of the client are successful. Therefore, the Login() method is invoked. Note that as the LoginWS does not have any APEs responsible for authorisation, there is no need for an ADC for the web service. Only the root collection’s ADC (ADC1) is invoked to combine decisions from APE1 and APE2.

Similarly, Figure 15 shows the sequence of authorisation steps involved in authorising a user to access the ReadContactDetails() method of the EssentialDetailsWS web service. EssentialDetailsWS uses the push-model of authorisation. Figure 16 shows the sequence of authorisation steps involved in authorising a user to access the ReadDetails() method of the ClinicDetailsWS web service. ClinicDetailsWS uses the pull-model of authorisation. Finally, Figure 17 shows the sequence of authorisation steps involved in authorising a user to access the ReadDetails() method of the HealthDetailsWS Web service. HealthDetailsWS uses the push-model of authorisation.

8.1.3 Discussion

1 We mentioned in Section 5.6 that the AZM in an organisation decides whether authorisation credentials and runtime attributes are pushed or pulled, when the combination-model algorithm is deployed. The AZM can arbitrarily decide whether the credentials and attributes required for WSC, web service or web service method level APEs must be pushed by the CP or pulled by the authorisation components. For instance, the AZM may decide that the credentials from CCA1 (signed IP address) and the attributes from DAS1 (location) be pushed by the CP for verification by APE1. Similarly she/he may decide that the credentials and attributes required by APE3 must be pulled by the authorisation components. Another possibility is that the credentials may be required to be pushed by the CP and attributes pulled by an authorisation component.

2 In the above example, CCA2 is used by multiple APEs. If the push-model algorithm is deployed, the CP needs to fetch the Role Membership Certificate (RMC) only once and attach it to the SOAP-Authorisation header and need not fetch it multiple times for APE2, APE3 and APE4. The AZS sends the same RMC credential multiple times to the appropriate APEs when required.

3 WSC, web services or their methods need not always have an APE associated with them. They may rely on ancestor WSCs’ APEs to authorise requests to them. For instance, LoginWS and its Login() method in the above application do not have an APE associated with them. They rely on APE1 and APE2 for authorisation. We could also have the ‘Security’ WSC without an APE responsible for it. Similarly, an APE may give out authorisation decisions without making use of a CCA or a DAS. For instance, APE2 need not rely on CCA2 or any other CCA or DAS for making authorisation decisions.
Figure 14  Authorisation steps involved in invoking Login() method on LoginWS
**Figure 15** Authorisation steps involved in invoking ReadContactDetails() method on EssentialDetailsWS
Figure 16 Authorisation steps involved in invoking ReadDetails() method on ClinicDetailsWS
### Figure 17: Authorisation steps involved in invoking ReadDetails() method on HealthDetailsWS

**Consultation**

<table>
<thead>
<tr>
<th>Complaint</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Severe chest pain</td>
<td></td>
</tr>
</tbody>
</table>

**Management Plan**

<table>
<thead>
<tr>
<th>Procedure</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Order of three</td>
<td></td>
</tr>
</tbody>
</table>

**Prescriptions**

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Diabetes, Metabolic Diseases</td>
</tr>
</tbody>
</table>

**Preventive advice**

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No diet advice</td>
</tr>
</tbody>
</table>

**Treatment plan**

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Nuan of diabetes</td>
</tr>
</tbody>
</table>

**Costs and options**

- Will need $500 for procedure, will accept Medicare (MDM) rebate

**Authorisation steps when HealthDetailsWS.ReadDetails() method is invoked**

- Client Group is involved
- Consent is obtained
- ID is verified
- Type is verified
- User is connected
- User access is verified
- Access is involved
- Service is involved
- Method is involved
- Web service method is involved
8.2 Performance evaluation

We implemented the healthcare application for a medical practice on a Windows 2003 Server running on a P4 machine with 512 MB RAM. The WSAA architecture introduces an average of 100 milliseconds (ms) delay when invoking the Login() method on LoginWS. Without the architecture deployed (Figure 18), it takes 15 milliseconds (ms) to get a response from the LoginWS. With the WSAA architecture deployed (Figure 19), it takes 109 ms. Delay in this case is 94 ms. Similarly, the WSAA architecture introduces an average of 100 ms delay when the ReadContactDetails() method is invoked on EssentialDetailsWS. Without the WSAA architecture deployed (Figure 20), it takes 2412 ms to get a response from the EssentialDetailsWS. With the WSAA deployed (Figure 21), it takes 2531 ms. Delay in this case is 119 ms. This once again shows that the WSAA architecture only causes an average of about 100 ms performance delay. This is reasonable as the benefits of our architecture outweigh this little overhead.

8.2.1 Discussion

In our architecture, coarse-grained WSC level policies are first evaluated and only if the authorisation is positive, finer-grained Web Service (WS) and web service method level policies are evaluated. This also increases the performance of our architecture. For instance, if APE1 (for root WSC, medical practice, in Figure 10) gives a negative authorisation, no further APEs are evaluated. However, all the credentials and attributes required for authorisation (informed by WS-AuthorisationPolicy attached to WSDL statement) are sent by the CP to the security manager when the request is made to the web service. We made this design choice to reduce the number of messages from the CP to the security manager. However, in the case of pull-model, the AZS (AZS) only pulls all the required credentials from CCAs only for one APE, when necessary, and sends them to the appropriate WSC, WS or WS method level APE. The APE itself pulls the required attributes from the corresponding DAS in the pull-model. Refer to the system sequence diagram in Section 5.7. If the APE gives out negative authorisation, the AZS need not pull further credentials required for other APEs.

9 Benefits of the proposed architecture

Some of the key advantages of the proposed architecture are as follows:

1 **Support for multiple AC models:** The WSAA supports multiple AC models including MAC, DAC and RBAC models. The access policy requirements for each model can be specified using its own policy language. The policies used for authorisation can be fine-grained or coarse-grained depending on the web service requirements. AC mechanisms can either use the push-model or pull-model or even a combination of both for collecting authorisation credentials on behalf of the client.

2 **Support for legacy applications and new web service-based applications:** Existing legacy application systems can still function and use their current AC mechanisms when they are exposed as web services to enable an interoperable heterogeneous environment. Once again, various access policy languages can be used to specify the AC rules for users. They could adopt a push or a pull-model for collecting credentials. At the same time, the WSAA supports new web service-based
applications built to leverage the benefits offered by the SOA. New access control mechanisms can be implemented and used by both legacy and new web service applications. A new AC mechanism can itself be implemented as a web service. All the WSAA requires an end-point URL and interface for the mechanism’s APE.

3 **Decentralised and distributed architecture:** A web service can have one or more responsible APEs involved (each with its own end-point defined) in making the authorisation decision. The APEs themselves can be web services specialising in authorisation. This feature allows the WSAA to be decentralised and distributed. Distributed authorisation architecture, such as ours provide many advantages such as fault tolerance and better scalability and outweighs its disadvantages such as more complexity and communication overhead.

4 **Flexibility in management and administration:** Using the hierarchy approach of managing web services and collections of web services, authorisation policies can be specified at each level making it convenient for WCM and WSM to manage the entities they are responsible for.

5 **Ease of integration into platforms:** Each of the components involved both in the administration and runtime domains is fairly generic and can be implemented in any middleware including the .NET platform as well as Java-based platforms. The administration and runtime domain related APIs can be implemented in any of the available middleware. We have implemented the WSAA within the .NET framework.

6 **Enhanced security:** In our architecture, every client request passes through the WSM and then gets authenticated and authorised. The security manager can be placed in a firewall zone, which enhances security of collections of web service objects placed behind an organisation’s firewall. This enables organisations to protect their web service-based applications from outside traffic. A firewall could be configured to accept and send only SOAP request messages with appropriate header and body to the responsible security manager to get authenticated and authorised.

10 **Concluding remarks**

We introduced the security issues for the SOA and then discussed the requirements for the design of an authorisation framework for the SOA. In particular, we distinguished the authorisation requirements for the web service layer and the business process layer comprising the SOA. In this paper, we proposed an authorisation architecture for the web services of the SOA. The WSAA extends the web services security layer. We also provided extensions to the messaging and description layers to support the proposed architecture. We described the functions of each of the components in the WSAA and introduced a new web services model. We then discussed how authorisation related objects can be managed in the WSAA as well as how authorisation policy evaluation is done. We discussed how runtime authorisation data is generated automatically by the CRM component. We defined three authorisation algorithms for push-model, pull-model and a combination-model that support various possibilities of collecting credentials required to authorise a client’s request. We discussed the support for management of web services and WSC. The proposed management features make it convenient for the managers to manage a set of possibly related web services.
Figure 18 LoginWS.Login() method response time without WSAA deployed
Figure 19  LoginWS.Login() method response time with WSAA deployed
Figure 20

EssentialDetailsWS.ReadContactDetails() method response time without WSAA deployed is: 2412 ms
Figure 21
EssentialDetailsWS.ReadContactDetails() method response time with WSAA deployed

EssentialDetailsWS.ReadContactDetails() method response time (with WSAA deployed) is: 2531 ms
We have implemented a prototype of the proposed architecture and integrated it within the .NET framework. We have demonstrated an application of our authorisation architecture for web services using a simple case study in the healthcare domain. The WSAA architecture introduces an average of 100 milliseconds performance delay when a web service method is invoked. We believe this is reasonable as the benefits of our architecture outweigh this little overhead.

The WSAA supports both legacy and new AC models and mechanisms; it supports legacy applications exposed as web services, as well as new web service-based applications built to leverage the benefits offered by web services; it is decentralised and distributed and provides flexible management and administration of web service objects and authorisation information. The proposed architecture is easy to integrate into existing platforms and provides enhanced security by protecting exposed web services from outside traffic.

As a part of our future work, we intend to propose a Business Process Authorisation Architecture (BPAA) that leverages the features provided by the WSAA, to provide authorisation services for the business processes layer of the SOA (see Figure 1). The BPAA will provide orchestration services to coordinate the authorisation decisions from individual partners’ APE involved in a business process workflow, where each partner web service can potentially be implemented by different organisations in a heterogeneous environment. The BPAA will extend the business process security layer of the SOA. The WSAA and the BPAA together, will provide a comprehensive authorisation framework for the SOA.

Acknowledgements

The author Vijay Varadharajan would like to thank Microsoft Research Asia and Australian Research Council for supporting this research work.

References


