

## Securing SOAP E-Services\*

E. Damiani<sup>1</sup>, S. De Capitani di Vimercati<sup>2</sup>, S. Paraboschi<sup>3</sup>, P. Samarati<sup>1</sup>

<sup>1</sup> Dipartimento di Tecnologie dell'Informazione  
Università di Milano  
Via Bramante 65  
26013 Crema, Italy  
{damiani, samarati}@dti.unimi.it

<sup>2</sup> Dipartimento di Elettronica per l'Automazione  
Università di Brescia  
Via Branze 38  
25123 Brescia, Italy  
decapita@ing.unibs.it

<sup>3</sup> Dipartimento di Elettronica e Informazione  
Politecnico di Milano  
Piazza L. da Vinci 32  
20133 Milano, Italy  
parabosc@elet.polimi.it

Received: date / Revised version: date

**Abstract** Remote service invocation via HTTP and XML promises to become an important component of the Internet infrastructure. Work is ongoing in the W3C XML Protocol Working Group to define a common standard, and solutions like SOAP and XML-RPC are already used in a few situations, demonstrating the potential. However, no standard technique for access control security is currently defined for these protocols. In this paper, we propose an approach that relies on the XML structure of SOAP requests to support fine-grained authorizations at the level of individual XML elements and attributes that compose a SOAP call. The result is a simple yet general technique to specify and enforce fine-grained access control for e-services.

**Key words** SOAP – e-services – access control – digital certificates

---

\* A preliminary version of this paper appeared under the title "Fine-Grained Access Control to SOAP E-Services" in the *Proc. of the 10th World Wide Web Conference*, May 2001, Hong Kong.

## 1 Introduction

E-services will become a central component in the Web infrastructure, as they will be the main tool for the realization of a strict integration between the information systems of modern organizations [7]. While many autonomous e-services have been successfully designed and implemented, a still open issue is how multiple e-services can be seamlessly integrated while preserving (and improving) the high availability and security provided by current Web technology. The Hypertext Transfer Protocol (HTTP), the most common protocol used on the Web today, has proved to be an effective, scalable technology for transferring multimedia information, but it was not designed for accessing distributed e-services. Indeed, calls to e-services are more easily modeled by *distributed object protocols* as *Remote Method Calls (RMCs)*, such as *CORBA* [24], *DCOM* [2], and *Java-RMI* [11], in which clients pass parameters to remote components and get some kind of result in return. Many RMC-based protocols support object invocation access policies that govern whether a client, acting on behalf of its current user, can invoke the requested operation on a target object. The policy is normally enforced by the software implementing the protocol. Early work was made with reference to remote procedure calls in the framework of the *Distributed Computing Environment (DCE)* [15,25]. Currently, distributed object-oriented architectures include full support for access control. For instance, *CORBA Security Service* access policies [19] are expressed in terms of the user's privilege attributes, which are encapsulated in a *Credentials* object, and the target object's control attributes, which are encapsulated in an *AccessDecision* object. The control attributes, which are associated with every object that accepts invocations, describe the authorizations that a user must have in order to be allowed to invoke the requested operation on the specific object implementation. Most object invocation access control services are implemented by intercepting all invocations, possibly on both client and server sides. Having intercepted the object invocation, the *Credentials* object is consulted to obtain the client's authorizations, which are then passed as a parameter to the *AccessDecision* object. Finally, *AccessDecision* either grants or denies permission to continue the object invocation. Access control provisions like those provided by *CORBA Security Services* would be particularly useful on the public Internet; on the other hand, other features of RMC-based distributed object protocols have proved to be rather unsuitable for Internet use. These protocols exhibit two main problems that prevent their large-scale use on the Net.

- *High bandwidth and low latency requirements*: Many RMC-based protocols require considerable bandwidth due to their high service to data packets ratio. For instance, *DCOM's ping-based* lifecycle management requires continuous conversation between client and server to keep the interaction alive.
- *Firewall traversal and user authentication*: Many organizations are reluctant to enable RMC-based protocols such as *CORBA-IIOP* or *DCOM* through their security firewalls. Moreover, current authentication techniques for RMC protocols were not designed for Internet use. For instance, in the case of *DCOM*, it seems unlikely that generic Internet-based clients will ever be able to perform domain-based authentication with *DCOM* application servers.

Several vendors provide patches based on *tunneling firewall-traversal*, encapsulating packets sent to RMC servers into a single packet stream to a *bastion host*. However, these products tend to be very sensitive to configuration mistakes and are not interoperable. To foster a clean solution to this problem, the Internet and Web communities have provided several proposals for the use of XML in *lightweight* network protocols and distributed applications: XML-RPC [32], SOAP [30], XMI [31], WebDAV [26], ICE [29], and IOTP [6] are only a few examples. In this paper, we focus on SOAP [30], seen as an attempt to codify the main concepts behind lightweight techniques into a simple and generic protocol that can serve as an industry standard. The XML Protocol Working Group of the W3C (<http://www.w3.org/2000/xml>) is studying the definition of protocols for the remote invocation of services in the XML context and is currently dedicating considerable attention to SOAP. SOAP is a clean and elegant solution to the problems above-mentioned; however, no standard technique for access control security has been yet defined for either HTTP or SOAP itself. Concerns have been raised about the possibility that different SOAP applications will deal with embedded security in different ways, leading to application-dependent security holes. We propose a simple yet general technique to specify and enforce *fine-grained access control* for SOAP-like invocations, leveraging the XML encoding used by SOAP for both service invocation and responses. Our approach allows the specification of *fine-grained usage policies* for e-services using XML and is based on an extension and adaptation of our previous proposal for regulating access to XML documents [3,4]. We believe the application of access control techniques to XML-based service invocations to be crucial for the development of reliable and secure e-services.

The paper is structured as follows. Section 2 presents the features of the SOAP protocol that are relevant to our approach. Section 3 discusses the limitations of firewall-based security policies applied to SOAP. Section 4 presents the basic idea of our proposal for a fine-grained flexible access control technique for SOAP invocations, which overcomes dependence from firewalls. Section 5 how subjects submitting service requests are characterized via SOAP headers. Section 6 describes the security model regulating service accessibility and Section 7 describes how access requests are filtered based on the specified authorization. Section 8 discusses the design and implementation of an *Authorization Filter* enforcing our approach. Finally, Section 9 draws our conclusions. The paper also contains an appendix reporting the XML schemas of the requester header and of the authorization syntax.

## 2 SOAP in a nutshell

We give a brief and informal overview of the SOAP protocol, including the format of SOAP HTTP requests and responses. For the sake of conciseness, we only deal with SOAP features that are relevant to our approach, and therefore our description does not attempt to be exhaustive.

SOAP requests carry remote method invocations over HTTP. They are fully declarative, inasmuch they do not dictate how the target component should han-

```

POST /QuoteService HTTP/1.1
SOAPAction="http://www.acme.com/Quote"
Content-Type: text/xml; charset="UTF-8"
Content-Length: nnnn
<!-- XML tree encoding the invocation goes here -->

```

**Fig. 1** A HTTP header carrying a SOAP request

the request. Client applications could freely integrate this service with others, using HTTP as the common transport protocol. The HTTP header associated with a SOAP request, describes the fact that the message carries a SOAP action. As an example, the HTTP POST request in Figure 1 encodes the invocation of a quote service using the SOAP protocol. It should be noted that while the HTTP request in Figure 1 points to a valid *Uniform Resource Identifier* (URI) of `http://www.acme.com/GetQuote`, it leaves it entirely to the *SOAP software gateway* behind the URI to decide how to activate the corresponding local component and invoke locally the specified method. Note that the `Content-Type` header contains the generic value `text/xml`, used by all XML-based HTTP traffic. In principle, the `SOAPAction` field could be used by firewalls that, by looking at the URI value of the field, could filter out all HTTP requests carrying SOAP traffic, or, conceivably, allow only certain interfaces to pass through. However, this would require the firewall to be SOAP-aware to some extent.

A SOAP message contains the encoded invocation; its lexicon is defined by a standard XML namespace `env`. The SOAP message is used mainly to encode parameters' datatypes in a platform independent way, much as CORBA's *Common Data Representation* [24] or DCOM's *Network Data Representation* [2]. The SOAP message includes a root `Envelope` element with two children: `Header`, which is optional, and `Body`, as follows.

- `Envelope` provides the serialization context for the method calls that follow and can contain additional attributes (qualified by a suitable XML namespace). Elements encoding methods and parameters must also be namespace-qualified.
- `Body` contains a first sub-element whose name is the method name. This element should contain all the information that the software gateway needs to perform the corresponding local invocation. Namely, it contains a child element for each parameter. As an example, method `GetQuote` in the SOAP request in Figure 2(a) contains the different parameters characterizing a quote service request, like the ZIP code of the origin and destination points and the type of service.
- `Header` contains auxiliary information (called *header blocks*) not functionally related to the method invocation, such as transaction management and payment. SOAP headers may contain the standard `actor` and `mustUnderstand` attributes (as well as other optional, namespace-qualified ones), respectively stating the URI of the final destination of the message and whether header processing capability on the part of the recipient is mandatory (1) or not (0). In the remainder of the paper, we shall use SOAP header blocks to support

```

<env:Envelope
  xmlns:env = "http://www.w3.org/2001/06/soap-envelope"
  xmlns:acme="http://www.acme.com/soap"
  env:encodingStyle = "http://www.w3.org/2001/06/soap-encoding">
  <env:Header acme:id="ref-0"> <!-- header blocks go here --> </env:Header>
  <env:Body>
    <acme:GetQuote acme:id="ref-1">
      <acme:OriginZIP> 90070 </acme:OriginZIP>
      <acme:DestZIP> 16804 </acme:DestZIP>
      <acme:Weight> .500 </acme:Weight>
      <acme:ServiceType> Overnight </acme:ServiceType>
    </acme:GetQuote>
  </env:Body>
</env:Envelope>

```

(a)

```

HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
<env:Envelope
  xmlns:env = "http://www.w3.org/2001/06/soap-envelope"
  xmlns:acme="http://www.acme.com/soap"
  env:encodingStyle = "http://www.w3.org/2001/06/soap-encoding">
  <env:Body>
    <acme:GetQuoteResponse>
      <acme:Amount> 18 </acme:Amount>
    </acme:GetQuoteResponse>
  </env:Body>
</env:Envelope>

```

(b)

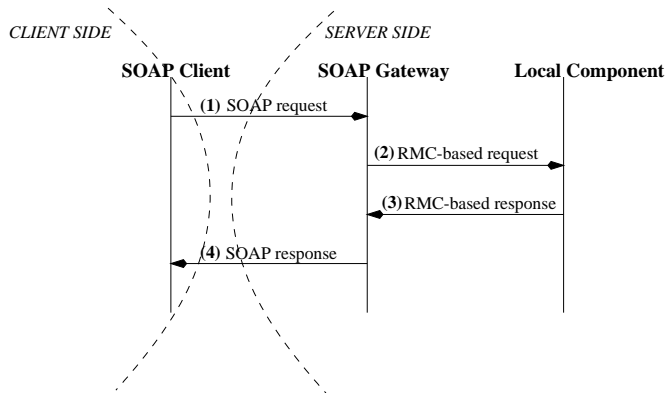
**Fig. 2** An example of a SOAP request (a) and the corresponding response (b)

client identification and a variety of security arrangements in the framework of a complete access control technique.

A SOAP response is similar to a request, apart from the fact that it adds a Response suffix to the element name used for the method. For instance, for the method `GetQuote`, the response element is `GetQuoteResponse`. A sample response to our `GetQuote` call of Figure 2(a) is reported in Figure 2(b).

### 3 Firewall impact on object protocols and SOAP

Most organizations insert *firewalls* between their publicly accessible Web servers and the remote clients who can access those servers, blocking incoming traffic according to various criteria. Firewalls partition the Internet in *security cells*, whose boundaries are difficult to penetrate for remote service invocations. As a simple



**Fig. 3** Execution Sequence of a SOAP Call

example, consider the TCP/IP architecture, where each *well-known service* is assigned a *port number* and each service request carries that number. While blocking all ports except the standard port 80 used for HTTP connections is a common practice, it prevents using distributed object protocols like CORBA or DCOM, which rely on dynamically assigned ports for remote method invocations. Enabling access to an e-service through a firewall requires manual intervention for the firewall configuration, discouraging free development of applications using e-services as building blocks. Furthermore, clients of distributed applications that lie behind another corporate firewall suffer a similar problem. Requiring clients to reconfigure their firewalls to access a remote e-service is a rather unrealistic assumption.

Several palliative remedies to these problems have been proposed in recent years, such as *COM Internet Services* (CIS) [14] and *Remote Data Services* (RDS) [21]. CIS makes it possible to use DCOM over HTTP on port 80, thanks to a special HTTP-based handshake used to establish the initial connection between client and server. From that point on, CIS relies on DCOM over TCP. While CIS does solve the firewall traversal problem, it is a platform-specific solution for Windows-based systems and retains the high bandwidth requirement of the original DCOM protocol. Remote Data Services (RDS) allows for instantiating remote DCOM objects and invoking their methods over HTTP. Again, however, RDS is platform-specific, as it relies on a proprietary Dynamic Link Library running on Microsoft Internet Information Service.

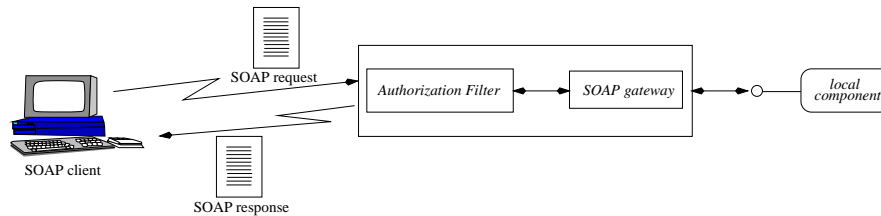
The SOAP messaging protocol piggybacks a lightweight distributed object protocol on top of HTTP, using HTTP connections to carry messages formatted with XML. In other words, SOAP defines a mechanism to pass commands and parameters between HTTP clients and servers that uses XML for data encoding and is therefore independent from the operating system, programming language, or object model used on either the server or the client side. SOAP solves the firewall traversal problem, as it relies on HTTP as the transport mechanism. Moreover, custom HTTP headers have been proposed as a technique for identifying SOAP invocations [30]. As illustrated in Figure 3, when a client sends a SOAP invocation over HTTP, the header triggers the execution of a software gateway on the server side,

which forwards the invocation to the target component using (locally) a suitable heavyweight RMC-based protocol. This technique addresses the bandwidth problem, as it uses a simple XML structure for both invocation and response. Moreover, relying on custom HTTP headers makes it possible for system administrators to configure firewalls to selectively block out SOAP requests using SOAP-specific HTTP headers. Besides the firewall security benefits of designing SOAP using extended HTTP headers, the SOAP specification does not define any protocol-specific security features. SOAP implementations may well utilize any standard HTTP security feature, taking advantage of HTTP authentication mechanisms as well as *SSL* for secure channel communications (using secure HTTP connections via *HTTPS*). Also, *secure cookies* have been proposed [20] to provide user authentication, integrity and confidentiality when interacting with WWW sites. However, these techniques are not a valid substitute for a fully-fledged security model, and several security issues related to SOAP are still to be solved. Indeed, both firewall filtering of HTTP headers and the secure cookies approach have been conceived for simple document retrieval on the WWW and cannot be considered satisfactory for remote invocations. Access to e-services requires a more sophisticated security model than the one normally applied to HTTP traffic.

#### 4 Adding fine-grained security regulations to SOAP

While the invocation of Figure 2(a) does not contain any provision for access control, the organization managing a remote interface to an international courier service is likely to impose some constraints. For instance, getting quotes could be restricted to retailers, or allowed only for customers connecting from a trusted domain. However, besides the firewall security benefits of designing SOAP using extended HTTP headers, the SOAP specification does not define any protocol-specific security features. Of course, checking the first constraint could be done by writing application code on the server, while the second constraint could be enforced via the SOAP specific firewall configuration just discussed. Both these ad-hoc solutions, however, potentially restore the dependence on firewall configuration and operation, whose elimination was the reason why a lightweight, XML-based protocol like SOAP was selected in the first place. Our research line is to avoid firewall dependence while providing organizations managing e-service with full control on how their SOAP servers are used. To this end, we propose to employ *fine-grained XML access control techniques* to specify usage policies for SOAP based e-services, in order to obtain the full functionality of an object invocation access control service. Our approach is based on the assumption that the communication channel is *secure*. This is the only constraint that we set, but the satisfaction of this requirement is also relatively easy, as there are available many robust implementation of secure transport protocols, like *SSL* [8], that can be easily integrated into the components of the architecture. Another possibility is to encrypt XML content according to the *XML Encryption* specifications [28].

As illustrated in Figure 4, our access control system relies on an Authorization Filter which intercepts every invocation addressed to the SOAP gateway and



**Fig. 4** The System Operation

evaluates it against authorizations specifying restrictions to service accessibility. Based on the authorizations, the request may: be rejected; be allowed as is; or be filtered and executed in a modified form, where filtering of a request may involve elimination of some of its parameters that the current invoker is not allowed to specify. Once filtered, requests are passed to the SOAP gateway, which will produce a response to be returned to the client. The response also is sent through the access control system and may be subject to some filtering. In the sequel, we focus on the specification and enforcement of restrictions applicable to requests (i.e., request filtering). Response filtering can be performed in a similar way.

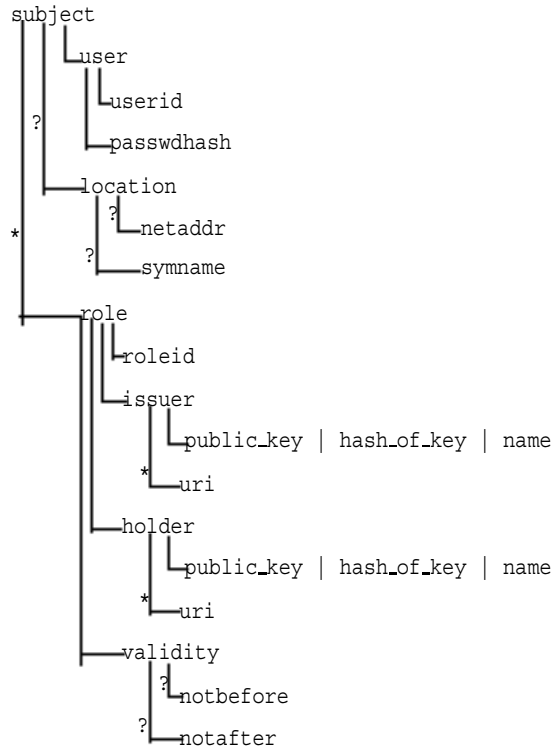
## 5 Service requesters

Before introducing our access control model, we need to characterize subjects submitting requests for services. We assume subjects to be characterized by:

- the *identity* of the user on behalf of which the client is executing;
- the *location*, either numerical or symbolic, from which the request originates;
- optional *roles* in virtue of which a user is presenting his request.

Intuitively, roles define privileged positions that the user can exercise, such as, *premier\_member* or *associate*. The support of roles appears crucial in the open context in which SOAP can operate, where not all *individuals* may be registered at a service and requests may arrive from previously unknown parties. In such a context the ability to invoke a service often depends on the capacity in which a user can exercise [1,12] (which we characterize with a *role*) rather than on who the user actually is. Because of the open and dynamic nature of the system, we assume role activation not be controlled by the server managing SOAP services itself (as usually done in centralized frameworks [22]). Rather, role enabling is enforced by digital certificates that can be associated with requests and that prove that the holder enjoys some roles.

Our approach exploits the features of the SOAP protocol, and represents all the information characterizing the subject by means of a custom header included in each SOAP call. Of course, the use of this feature is not mandatory: the system may also rely on usual HTTP authentication techniques as in [5]. However, custom headers provide a uniform description framework for service requests and subjects submitting them, and represent an elegant and model-neutral way to specify security arrangements. The structure of the custom header for describing subjects



**Fig. 5** Structure of subject custom header

is illustrated in Figure 5, while the complete corresponding XML Schema is reported in Section A.1. The header has a root `subject` with three types of children: `user`, `location`, and `role`, the first of which is mandatory<sup>1</sup> and the remaining two are optional. Also, the latter can have multiple occurrences. Element `user` contains the identity (element `userid` and the hashed password `passwdhash`) of the user on behalf of whom the request is submitted. Element `location` defines the network address of the machine from which the request originates and can be symbolic (`symname`) or numeric (`netaddr`). Each element `role` carries a certificate enabling the user to exercise the stated role. We do not impose a specific XML structure for certificates, and different systems may use different representations. For the sake of concreteness, in this paper we refer to the sample representation illustrated in Figure 5 and XML Schema in Section A.1. In particular, we assume the certificate enabling a role to be characterized by four elements: `roleid`, `issuer`, `holder`, and `validity`. Element `roleid` states the identifier of the role enabled by the certificate. Element `issuer` specifies the authority that released the certificate. Element `holder` defines to whom the certificate is referred (and may refer to

<sup>1</sup> Systems non requesting user authentication will have a dummy `Anonymous` identifier and a dummy password hash for this element.

```

<SOAP-SBJ:subject xmlns:SOAP-SBJ = "http://www.xmlsec.org/subject">
  <SOAP-SBJ:user>
    <SOAP-SBJ:userid> Alice </SOAP-SBJ:userid>
    <SOAP-SBJ:passwdhash SOAP-SBJ:hash-alg="sha1">
      xxj31ZMTZzkVA
    </SOAP-SBJ:passwdhash>
  </SOAP-SBJ:user>
  <!-- roles and certificates follow -->
  <SOAP-SBJ:role>
    <!-- role identifier follows -->
    <SOAP-SBJ:roleid> acme_premier </SOAP-SBJ:roleid>
    <!-- certificate issuer info follows -->
    <SOAP-SBJ:issuer>
      <SOAP-SBJ:public-key>
        ...
      </SOAP-SBJ:public-key>
    </SOAP-SBJ:issuer>
    <!-- end of certificate issuer info, begins holder info -->
    <SOAP-SBJ:holder>
      <SOAP-SBJ:name> Alice </SOAP-SBJ:name>
    </SOAP-SBJ:holder>
    <!-- end of holder info, begins validity info -->
    <SOAP-SBJ:validity>
      <SOAP-SBJ:notbefore> 2001-06-22.12:00:00 </SOAP-SBJ:notbefore>
      <SOAP-SBJ:notafter> 2001-07-31.24:00:00 </SOAP-SBJ:notafter>
    </SOAP-SBJ:validity>
  </SOAP-SBJ:role>
</SOAP-SBJ:subject>

```

**Fig. 6** Sample header block for SOAP subject credential

its identity or key). Finally, element *validity* imposes constraints on the time in which the certificate is to be considered valid. An example of a header block for a SOAP subject credential (written on a custom namespace SOAP-SBJ) is illustrated in Figure 6.<sup>2</sup>

Client authentication by means of our XML credential can be performed exactly as it is done for standard certificates, taking advantage of challenge-response [9] and secure channel technology like SSL [8] when needed. We note that location addresses stated in the header could be trusted only within secure infrastructure [16] or checked using HTTP services. We do not discuss here on this matter as our focus is on authorization rather than on authentication issues, but simply note that while they may indeed turn useful in several cases, they should be used with care.

Having completely characterized our requests we can now proceed illustrating the security model regulating request filtering.

<sup>2</sup> This figure contains a slight abuse of notation as technically the name space declaration should be given in an ancestor element in order to allow *subject* to be part of its scope.

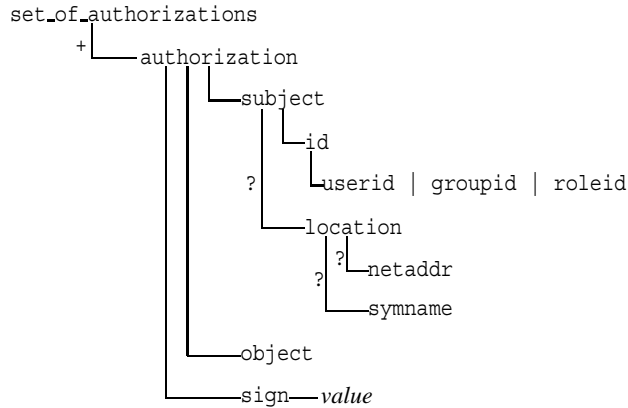


Fig. 7 Authorization structure

## 6 Security model

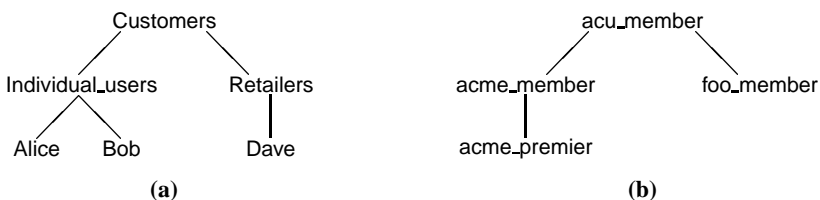
Request filtering is based on authorizations stating the format of requests that subjects can (or cannot) submit to the server. Each authorization is therefore characterized by (Figure 7):

- the description of the subjects to which it refers (*authorization subject*),
- the description of the requests to which it refers (*authorization object*),
- the statement of whether it states a permission or a denial for a given access.

Permissions and denials are characterized via a sign (“+” for permissions and “-” for denials). We now illustrate how authorization subjects and objects are characterized.

### 6.1 Authorization subjects

The subject element of an authorization defines to which subjects the authorization applies and can be specified with reference to any of the information on which requesters can be evaluated, namely: their user identifier, the roles they have enabled, or the location from which they are connected. Furthermore, we allow authorizations to be specified for groups, where a group is a named set of users [22]. The support of both groups and roles is not redundant, as the two exhibit different behaviors. In particular, unlike groups, roles carry a dynamic behavior and can be activated and deactivated by users at their will [22,23]. Also, we assume groups are maintained at the server while, as already discussed in Section 5, roles are not. Both groups and roles can be organized hierarchically. The group hierarchy reflects memberships of user/groups in other groups and, being groups not necessarily disjoint and possibly nested, defines a DAG. The role hierarchy reflects a specialization-generalization relationship among roles (e.g., `acme_member` and `acme_premier`). Locations also can be organized hierarchically, by using classical



**Fig. 8** An examples of group (a) and role (b) hierarchies

patterns to denote machines belonging to a given sub-network (e.g., 131.100.\* or \*.it), and resulting in a tree. Examples of group and role hierarchies are illustrated in Figure 8. Hierarchies are exploited in the specification of authorizations as authorizations specified for a non-minimal element apply to all elements below it the hierarchy. Intuitively, this corresponds to propagating the authorizations of a group to all its members, the authorizations of a role to all its specializations, and the authorizations for a location pattern to all the machines in its subnetwork.

Summarizing, we assume that the authorization subject refers to either a user, a group, or a role, and possibly to a location. As an example, authorization subject

```

<subject>
  <id> <groupid> Retailers </groupid> </id>
  <location> <netaddr> 131.175.* </netaddr> </location>
</subject>

```

defines an authorization applicable to all requests by users belonging to group Retailers and originating from machines in subnetwork 131.175.\*.

## 6.2 Authorization objects

The object element of an authorization defines to which requests the authorization applies and whether the requests (or portions of them) can, or cannot, be accepted. In our model, authorization objects are therefore *SOAP requests* themselves. This approach departs from traditional approaches (including the one on which our proposal is inspired [4]) characterized by a distinction between the object on which an access is required, and the action which is required on it. The explicit reference to requests allows to capture the broader concept of service, encompassing actions and objects on which they are executed; traditional authorization objects being just one of the parameters of the request [1].

To provide-fine grained specification, we allow specific elements/attributes of a SOAP request, such as the method name or the value of any of its parameters, to appear as authorization objects. Authorization objects therefore coincide with XML elements and attributes in the SOAP request, which we identify via *path expressions* written using a simple XPath-like syntax [27]. Intuitively, an *XPath expression* on an XML document tree is a sequence of element names or predefined functions separated by the character / (slash):  $l_1/l_2/\dots/l_n$ , which corresponds to a path on the XML document tree [4]. Path expressions may terminate with an

attribute name, syntactically distinguished prefixing it with the special character @. As an example, path expression `/env:Envelope/env:Body/acme:GetQuote` denotes the nodes of the `GetQuote` element (the method's name), which are children of `Body` elements, which in turn are children of `Envelope` elements. *Absolute* path expressions, prefixed by a slash character, start from the root of the document while *relative* ones, starting with an element, describe a path whose initial point is any element in the document.

The XPath syntax allows to specify conditions on any element of a path expression. Conditions are enclosed within square brackets and may operate on the “text” of elements (i.e., the character data in the elements) or on names and values of attributes. This feature results very convenient as it allows us to easily specify content-dependent authorizations. For instance, path expression `env:Envelope/env:Body/acme:PlaceOrder/[./acme:ServiceType="48-hours"]` identifies all orders of type “48-hours”.

The evaluation of a path expression identifies a set of nodes. Accordingly, the path expression in the object field of an authorization identifies all nodes (elements and attributes of the request) to which the authorization applies. Authorizations referred to the request as a whole, are characterized by the path expression identifying the root, `Envelope`, of the request tree. Negative authorizations referred to the root will imply a complete rejection of the request.

Note that the path expression in the authorization object can also refer to the header (`Header` element or its descendants) of the request tree. This feature can be useful, for instance, to deal with certificates that subjects are, or are not allowed, to submit. In particular, a negative authorization on a certificate will result in the removal of the certificate before forwarding the request to the SOAP gateway.

### 6.3 Authorization syntax and semantics

We have already anticipated the structure of authorizations supported by our model in Figure 7. The complete XML Schema for authorizations is reported in Section A.2. Each authorization is characterized by a triple (*subject*, *object*, *sign*), where *sign* has value “+” for positive authorizations and value “-” for negative authorizations. A positive authorization states that, for requesters described by *subject*, the request elements identified by *object* can pass through the authorization filter unaltered. A negative authorization states that, for requesters described by *subject*, the request elements identified by *object* must be eliminated by the authorization filter before forwarding the request to the SOAP gateway.

To illustrate an example of a security policy for e-services and its representation with our authorizations, consider an international courier service whose customers are split into two groups: `IndividualUsers` grouping private users, and `Retailers` (Figure 8(a)). Also, the service recognizes different privileged roles that users can enable by presenting appropriate credentials (Figure 8(b)). Examples of such roles are `acu_member`, identifying enrollment in a nation-wide association of courier users, and `acme_premier`, enabled by a credential released to preferred customers. Within this scenario, let us now illustrate a sample policy regulating

1	<pre> &lt;subject&gt; &lt;id&gt; &lt;groupid&gt; IndividualUsers &lt;/groupid&gt; &lt;/id&gt; &lt;/subject&gt; &lt;object&gt; /env:Envelope[env:Body/acme:PlaceOrder/ServiceType="48-hours"] &lt;/object&gt; &lt;sign value = "+" /&gt; </pre>
2	<pre> &lt;subject&gt; &lt;id&gt; &lt;groupid&gt; Retailers &lt;/groupid&gt; &lt;/id&gt;   &lt;location&gt; &lt;netaddr&gt; 131.175.* &lt;/netaddr&gt; &lt;/location&gt; &lt;/subject&gt; &lt;object&gt; /env:Envelope[env:Body/acme:PlaceOrder] &lt;/object&gt; &lt;sign value = "+" /&gt; </pre>
3	<pre> &lt;subject&gt; &lt;id&gt; &lt;roleid&gt; acu_member &lt;/roleid&gt; &lt;/id&gt; &lt;/subject&gt; &lt;object&gt; /env:Envelope[env:Body/acme:PlaceOrder] &lt;/object&gt; &lt;sign value = "+" /&gt; </pre>
4	<pre> &lt;subject&gt; &lt;id&gt; &lt;roleid&gt; acu_member &lt;/roleid&gt; &lt;/id&gt; &lt;/subject&gt; &lt;object&gt; /env:Envelope/env:Body/acme:PlaceOrder/acme:Corp_Discount_Code &lt;/object&gt; &lt;sign value = "-" /&gt; </pre>
5	<pre> &lt;subject&gt; &lt;id&gt; &lt;roleid&gt; acme_premier &lt;/roleid&gt; &lt;/id&gt; &lt;/subject&gt; &lt;object&gt; /env:Envelope/env:Body/acme:PlaceOrder/acme:Corp_Discount_Code &lt;/object&gt; &lt;sign value = "+" /&gt; </pre>

**Fig. 9** An example of authorizations regulating access to a delivery service

the service of automatically placing delivery orders via Internet and its representation as authorizations supported by our model (the authorizations are reported in Figure 9).

- *Individual users can place order for 48-hour service (quicker deliveries are not accepted through this interface).*

A positive authorization (1) on root Envelope is specified for group IndividualUsers with a condition evaluating the type of service. If the condition is not satisfied the authorization does not apply.

- *Retailers can place any order if connected from subnetwork 131.175.\**

A positive authorization (2) on root Envelope is specified for the group Retailers and location 131.175.\* and with a condition evaluating the name of the method to be PlaceOrder.

- *Requesters enjoying role acu\_member can place any order, but a corporate code for their orders can be submitted only if the requester enjoys the more privileged role of acme\_premier.*

A positive authorization (3) on root Envelope is specified for role acu\_member, with a condition evaluating the name of the method to be PlaceOrder.

A negative authorization (4) on element Corp\_Discount\_Code in place order requests is specified for role acu\_member.

A positive authorization (5) on element Corp\_Discount\_Code in place order requests is specified for role acme\_premier.

Negative authorizations therefore restrict the requests (or elements within) that subjects can submit. At this point one could object that negative authorizations granted to roles are unreliable, as, given that roles are not maintained at the service, a user associated with a role can avoid being subjected to some restrictions simply by not presenting the certificate for the role. While we acknowledge this fact (which requires careful authorization administration), we also notice that this is not a problem in our context where negative authorizations for a role will be primarily intended to specify exceptions to positive authorizations granted to the role itself, and such positive authorizations therefore apply if the corresponding cer-

tificate is indeed presented. The next section illustrates the access control process more in detail.

## 7 Access control

Our access control filter intercepts every SOAP request (and its response) and evaluates it against the specified authorizations. The authorization enforcement is based on a processing labeling the request tree according to the authorizations, in a way similar to that proposed in [4]. The first step of the process consists in determining all the authorizations applicable to the requests. These are authorizations whose subject matches with the subject header of the request. Once these authorizations are determined, the path expression in their object field is evaluated. According to the semantics of the authorizations, a positive (negative resp.) authorization implies a “+” (“-” resp.) on the attributes/elements identified by the path expression in the object field of the authorization. Of course, the case can be that multiple conflicting authorizations apply to the same node (element/attribute) of the request tree. In this case, priorities must be established to determine the unique sign to be considered for the node. In principle, different priority policies can be applied, and the system can be parametric with them [10,13]. To keep authorization administration simple and intuitive, we consider a specific priority, which appears natural for the specific context under consideration. Our priority policy obeys the following principles:

- *Specificity of the user-group hierarchy*: Authorizations specified for a user (or a sub-group) take precedence over authorizations specified for a group.
- *Specificity in the role hierarchy*: Authorizations specified for a role take precedence over authorizations specified for, direct or indirect, super-roles.
- *Priority of the “individual” over roles*: Authorizations specified for a user (either directly or as a member of a group) take precedence over authorizations specified for any of the roles the user has activated.
- *Maximum privilege*: If a request has multiple roles enabled, the union of their privileges is taken; as it is customary in corresponding paper-world situations. Consequently, in the case where some roles hold contradicting authorizations not resolved by the most specific priority (namely the roles are not in a hierarchical relationship), the positive authorization takes precedence.

Note that last two principles are consistent with the fact that roles are managed through certificates and therefore a user can avoid submitting a role if a negative authorization applies to it, as noted in the previous section.

Once the initial labeling has been obtained, labels are propagated down the tree with a semantics that authorizations on an elements are considered to apply not only to the element node but to the subtree rooted at it. Note that if no label is specified for the root, or if the root has a “-” label, there is no point in performing propagation and the request can be *rejected* immediately. This decision assumes a closed policy and is consistent with our assumption of using negative authorizations as exceptions to permissions otherwise stated. (An open policy can

be modeled in such a framework simply by specifying a single authorization on the envelope of any request and applicable to every subject). Assuming an existent positive labeling of the root, propagation can then be enforced to determine how the request should be filtered. The propagation of labels obeys the principle:

- *Specificity in the request tree*: Authorizations specified for a node take precedence over authorizations inherited from its ancestors.

After propagation, each node in the request tree is associated with a sign, “+” or “–”, stating the access control outcome with respect to the node. Accordingly the request must be filtered by removing all those nodes labeled with “–”, as well as the subtrees rooted at them, which would otherwise remain pending.<sup>3</sup>

Summarizing, depending on the authorizations to be enforced, each request can:

- *pass unaltered*: after propagation all nodes have a “+” label. The request is forwarded to the SOAP gateway as it was submitted.
- *be rejected completely*: the root of the request tree has a “–” or no label. The request is blocked and not forwarded to the SOAP gateway.
- *pass modified*: the root of the request tree has a “+” label but some nodes in the tree are labeled “–”. All subtrees rooted at a node negatively labelled are removed and the request so modified is forwarded to the SOAP gateway.

To illustrate consider the request in Figure 10(a) and assume authorizations to be as in Figure 7. Authorizations 1, 3, and 4 apply to the request. Authorization 1 does not produce any labeling as its path expression does not identify any element in the tree. Authorization 3 produces a “+” for node Envelope. Authorization 4 produces a “–” on Corp\_Discount\_Code. Their enforcement will therefore produce the filtered request obtained by pruning from the tree in Figure 10(a) the subtree rooted at Corp\_Discount\_Code, with the result shown in Figure 10(b). This filtered request will therefore be forwarded to the SOAP gateway.

## 8 Design and implementation

The authorization filter discussed in this paper is currently under development. In this section we discuss the main features of our design. Overall, the architecture of a SOAP system consists of a client, a SOAP gateway and a communication channel. The client produces the SOAP request and receives a SOAP response; the SOAP gateway translates the SOAP request to a call to a local or remote server; the answer of the server is then translated back to the format of the SOAP response; the communication between the client and the gateway uses the HTTP protocol. In this architecture, shown in Figure 4, access control is enforced by an Authorization Filter, illustrated next.

---

<sup>3</sup> We note that this approach differs from the one taken in the work on which our proposal is based [4], where elements labeled “–” could have been maintained for the sake of preserving the document structure and reachability of descendant nodes the requester was entitled to see. Indeed, while preserving the existence of a negatively labelled node is applicable when computing the visible view on an XML document, it may not correctly reflect the specifications to be enforced in the context of request filtering.

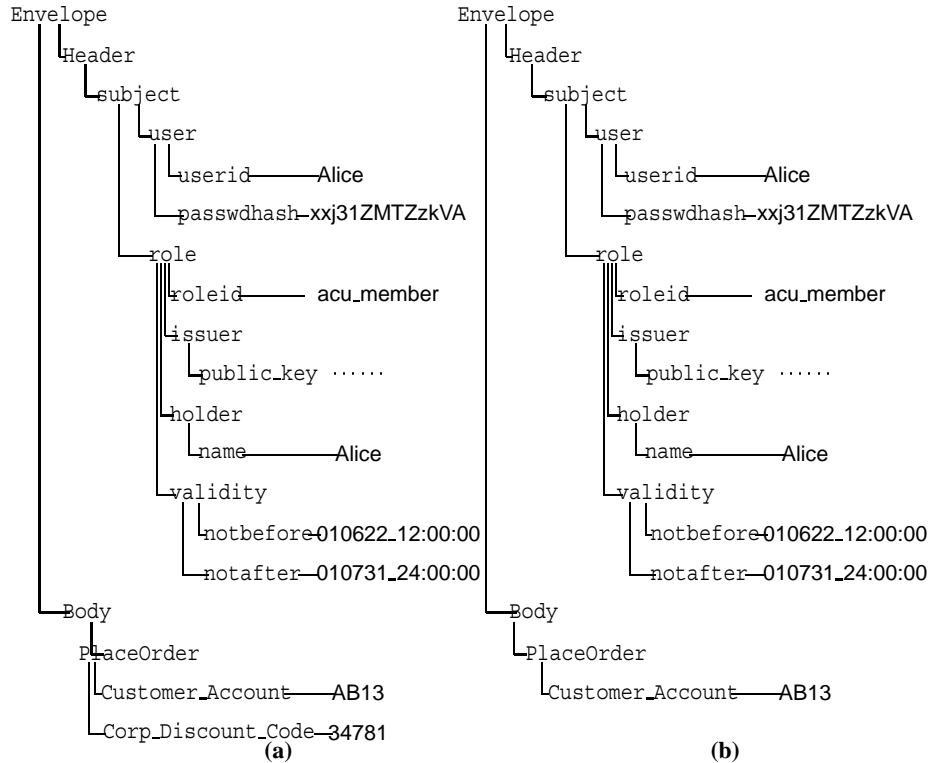
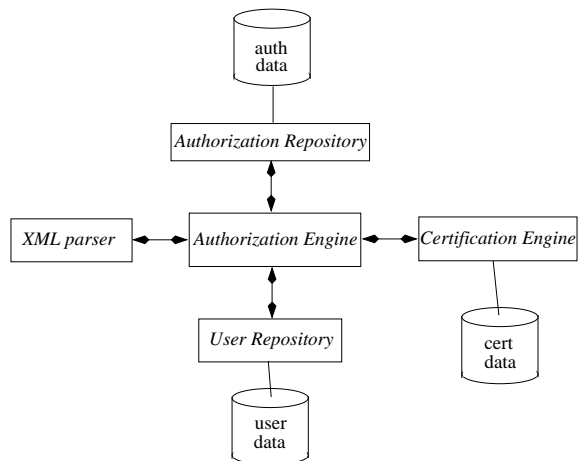


Fig. 10 An example of Request Filtering

### 8.1 Architecture of the Authorization Filter

The Authorization Filter is the core of the system. It is located on the communication channel between the client and the SOAP gateway and analyzes all the requests. Its internal architecture, illustrated in Figure 11, comprises of a *User Repository* that describes the users, groups and roles on which authorizations are defined, an *Authorization Repository* that describes all the privileges that are granted to users/groups/roles, a *Certification Engine* that evaluates the correctness of the certificates provided with the request, and an *Authorization Engine* that applies the model we presented to a given request instance to determine if the request must be restricted by the filter or can pass unaltered through it. The conversion from the textual XML message to an internal representation is the responsibility of an *XML parser*. Depending on the degree of integration, the request can be forwarded to the SOAP gateway after it has been again serialized by the XML parser, or it can be directly passed in an internal format, such as DOM.

**User Repository** The *User Repository* maintains the description of the users, groups, and roles that have been defined on the system. This information is stored persistently in XML. The User Repository offers an interface that permits to access the properties of every component (like the hashed password, re-



**Fig. 11** Architecture of the Authorization Filter

quired for user authentication), and the service it most typically offers permits to determine if a given authorization is applicable to a certain user instance, characterized by a userid, an IP address, possibly a symbolic name, and a list of certified roles. This evaluation may need to use service *isSubjectMoreSpecific* to enforce the priority policy illustrated in Section 7.

There are two main implementation strategies for the User Repository: 1) it can be designed as a system that requires initialization, possibly running in a separate thread; 2) it can be a stateless set of services available to the other components. Our current implementation choice opted for the first solution, which, although more difficult to manage and requiring a relative expensive initialization, also offers better performance. As a matter of fact, it analyzes the content of the stored repository only at startup and it can also use optimized data structures that permit to offer a low service time. The second solution instead would require to load the content of the Repository every time a request arrives.

**Authorization Repository** The *Authorization Repository* maintains the collection of authorizations that describe the security policy. The authorizations are persistently stored in XML format. The services offered by the Authorization Repository permit the retrieval of the authorizations applicable to a given request. A problem to be considered in this context is the organization to use for the data. A trivial solution consists in keeping all the authorizations in a single document, as a flat list. We selected a more sophisticated solution that stores the authorizations of each interface (characterized by the URI of the HTTP POST action) in separate documents. We also plan to investigate authorization indexing techniques, organizing the authorizations depending on their content, for example, object (i.e., ACLs) or subject (i.e., capabilities).

**Certification Engine** The goal of the *Certification Engine* is to evaluate the correctness of the certificates that a subject may present to substantiate his ability

to play a certain role. We assume that this service can be realized by the integration into our architecture of existing solutions [18] to the management of certificates, some of which are currently available under the Java platform. Such components support the integration of our service into a public-key infrastructure [17], realize sophisticated protocols for the exchange of a set of messages, and manage challenge-response authentications.

**Authorization Engine** The *Authorization Engine* is the main component of the Authorization Filter and it coordinates the use of the other subsystems. It reacts to the receipt of a request and parses its <Header> element to determine the subject. If the subject presents a userid and a password, it asks to the User Repository the hashed password of the user; if the comparison is successful, the user is authenticated. If the subject presents credentials, each one of them is verified with the Certification Engine. For every verified certificate, the corresponding role is associated with the subject. Then, the Authorization Engine retrieves all the authorizations applicable to requester asking the User Repository to verify if its subject corresponds to the actual subject submitting the request. For all the applicable authorizations, the path expression is evaluated and the Authorization Engine produces a labeling of the internal DOM representation of the SOAP request. Finally, the request that remains after the removal of the nodes with a negative label is passed to the SOAP gateway.

**XML parser** The XML parser is the component responsible for the conversion from the textual representation of XML to its equivalent memory description. A technical aspect that should be considered is the type of parser to use. Building on our previous experience on developing a related authorization system for selective access to XML data (<http://seclab.crema.unimi.it/~xml-sec>), our design exploits the DOM representation of the SOAP call. The reason for using the DOM representation is that, in our model, authorization objects are defined via generic expression of the XPath language, and the evaluation of these expressions may require to navigate the XML structure in arbitrary ways.

We note, however, that the construction of the DOM tree involves building a complete memory representation of the XML information and in some contexts (where the XML data require a considerable area of memory) this may become a bottleneck on the workload of the authorization system.

The alternative to the construction of a DOM representation is the use of a SAX parser, which analyzes the textual representation of the XML information and produces events for every component of the structure which emerges from the analysis. The SAX parser is indeed the mechanism upon which the construction of the DOM representation is based, and it is an efficient mechanism for the analysis of the XML content. The Authorization Filter can use a SAX parser to analyze an incoming stream of data, triggering *events* representing only the nodes that are authorized to be part of the request. This mechanism stores in memory only the status of the parser, consisting of the path connecting the current node with the root. In order to be evaluated in this context, the path expressions appearing as authorization object must satisfy a set of restrictions. The main restriction is that the path expression should be re-

stricted to descending terms and to conditions at the local level. This approach also benefits from a careful preprocessing of the authorizations.

The use of a SAX parser is particularly relevant for future management of authorizations on the response, which can often require the retrieval of a large amount of XML information; instead, it can be considered as less critical for the management of SOAP requests, where typically the size of the message is limited. Also, the SOAP gateway, for all the open source prototypes that we analyzed, manages the requests creating a DOM representation. If the Authorization Filter is realized with a strict integration with the SOAP gateway, passing to it directly the DOM representation of the request, the cost of the DOM conversion can be factored out.

### *8.2 Impact on current SOAP components*

The realization of the authorization services has a limited impact on the current components of the SOAP infrastructure. Indeed, the SOAP gateway does not have to be modified in a significant way because of the presence of the Authorization Filter. A solution, which strictly integrates the Authorization Filter within the SOAP gateway, simply delegates to the Authorization Filter the task of acquiring the request, in place of the DOM parser. The Filter parses the request and returns it to the gateway after the authorizations have been applied. A solution with no modification to the SOAP gateway can also be realized, where the Authorization Filter is separated from the SOAP gateway and communicates with it using the HTTP protocol. This solution may be the only feasible when the SOAP gateway is available as a monolithic executable module.

The client must be enriched to submit a SOAP request with the identification of the user and its certificates. The addition of a header containing the userid and the hashed password requires a trivial extension of the client. Moreover, clients unaware of the access control facility can easily be handled by adding a default header to their calls. The management of certificates instead requires a more complex service, for which it is convenient to reuse already available solutions, for reasons analogous to those presented for the implementation of the Certification Engine.

## **9 Conclusions**

SOAP is a viable solution to the problems raised by the Internet use of RMC-based protocols. However, lack of standardization could lead to application-dependent security holes in the enforcement of access control policies for SOAP. In this paper we have presented a general fine-grained authorization model for controlling SOAP requests and sketched the architecture of the system implementing this approach. The approach provides flexibility as it is able to support a variety of protection requirements concisely. We are currently addressing some specific features of the SOAP standard, such as the exception responses that may be generated by the SOAP gateway. We plan to exploit this characteristic in order to produce a

<FAULT> header with a description of the security violation detected by the Authorization Filter, offering to the client an explanation of the reason why a request was not fulfilled as it was expected. In our design, this mechanism to notify authorization violations is a configuration parameter of the system.

## 10 Acknowledgments

This work was supported in part by the European Community within the FASTER Project in the Fifth (EC) Framework Programme under contract IST-1999-11791 and by the Italian MURST within the DATA-X project.

## References

1. P. Bonatti and P. Samarati. Regulating Service Access and Information Release on the Web. In *Proc. of the 7th ACM Conference on Computer and Communication Security*, pages 134–143, Athens, Greece, November 2000.
2. N. Brown and C. Kindel. Distributed Component Object Model Protocol – DCOM/1.0, January 1998. <http://www.globecom.net/ietf/draft/draft-brown-dcom-v1-spec-03.html>.
3. E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. Design and Implementation of an Access Control Processor for XML Documents. *Computer Networks*, 33(1-6):59–75, June 2000.
4. E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. Securing XML Documents. In *Proc. of the 2000 International Conference on Extending Database Technology (EDBT2000)*, pages 121–135, Konstanz, Germany, March 2000.
5. E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. XML Access Control Systems: A Component-Based Approach. In *Fourteenth Annual IFIP WG 11.3 Working Conference on Database Security*, Schoorl, The Netherlands, August 2000.
6. D.E. Eastlake. Internet Open Trading Protocol Version 2 Requirements, August 2001. <http://www.ietf.org/internet-drafts/draft-ietf-trade-iotp2-req-01.txt>.
7. S. Feldman. The Changing Face of E-Commerce: Extending the Boundaries of the Possible. *IEEE Internet Computing*, 4(3):82–83, May/June 2000.
8. A.O. Freier, P. Karlton, and P.C. Kocher. The SSL Protocol - Version 3.0, March 1996. <http://ftp.nectec.or.th/CIE/Topics/ssl-draft/INDEX.HTM>.
9. B. Gladman, C. Ellison, and N. Bohm. Digital Signatures, Certificates and Electronic Commerce. <http://citeseer.nj.nec.com/277887.html>, 1999.
10. S. Jajodia, P. Samarati, M.L. Sapino, and V.S. Subrahmanian. Flexible support for multiple access control policies. *ACM Transactions on Database Systems*, 26(2):1–47, June 2001.
11. Java Remote Method Invocation (RMI). <http://java.sun.com/j2se/1.3/docs/guide/rmi/index.html>.
12. J. Kahan. WDAI: A Simple World-Wide Web Distributed Authorization Infrastructure. *Computer Networks*, 31(11-16):1599–1609, May 1999.
13. M. Kudo and S. Hada. XML Document Security and e-Business applications. In *Proc. of the 7th ACM Conference on Computer and Communication Security*, pages 87–96, Athens, Greece, November 2000.
14. M. Levy. COM Internet Services, April 1999. <http://msdn.microsoft.com/library/backgrnd/html/CIS.htm>.

15. S. Lewontin and M.E. Zurko. The DCE Web Project: Providing Authorizations and other Distributed Services to the World-Wide Web. In *Proc. of the 2nd World Wide Web Conference*, October 1994. [http://archive.ncsa.uiuc.edu/SDG/IT94/Proceedings/Security/lewortin/Web\\_DCE\\_Conf\\_94.html](http://archive.ncsa.uiuc.edu/SDG/IT94/Proceedings/Security/lewortin/Web_DCE_Conf_94.html).
16. D. Massey and S. Rose. DNS Security Introduction and Requirements, June 2001. <http://www.ietf.org/internet-drafts/draft-ietf-dnsex-dnssec-intro-00.txt>.
17. U. Maurer. Modeling a Public Key Infrastructure. In *Proc. of the Fourth European Symposium on Research in Security and Privacy*, volume LNCS 1146, pages 325–350, Rome, Italy, September 1996.
18. P. Nikander and A. Karila. A Java Beans Component Architecture for Cryptographic Protocols. In *Proc. of the 7th Usenix Security Symposium*, San Antonio, Texas, January 1998. <http://www.tml.hut.fi/Research/TeSSA/Papers/Nikander-Karila/nikander-karila-98.html>.
19. Object Management Group. *The CORBA Security Service Specification*. <http://cgi.omg.org/cgi-bin/doc?formal/01-03-08>.
20. J. Paajarvi. *XML Encoding of SPKI Certificates*. Internet Draft.
21. Remote Data Service: A Web Data Access Feature, 2000. <http://www.microsoft.com/data/ado/rds>.
22. P. Samarati and S. De Capitani di Vimercati. Access control: Policies, models, and mechanisms. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design*, LNCS 2172. Springer-Verlag, 2001.
23. R. Sandhu, D. Ferraiolo, and R. Kuhn. The NIST Model for Role-Based Access Control: Towards A Unified Standard. In *Proc. of 5th ACM Workshop on Role-Based Access Control*, Technical University of Berlin, Berlin, Germany, July 2000.
24. The Common Object Request Broker: Architecture and Specification, Revision 2.1, August 1997. <ftp://ftp.omg.org/pub/docs/formal/97-09-01.pdf>.
25. V. Varadharajan, C. Crall, and J. Pato. Authorization in Enterprise wide Distributed Systems: Design and Application. In *Proc. of the 14th IEEE Computer Security Application Conference (ACSAC'98)*, December 1998.
26. E.J. Whitehead. World Wide Web Distributed Authoring and Versioning (WebDAV): An Introduction. *ACM StandardView*, 5(1):3–8, March 1997.
27. World Wide Web Consortium (W3C). *XML Path Language (XPath) Version 1.0*, November 1999. <http://www.w3.org/TR/xpath>.
28. World Wide Web Consortium (W3C). *XML Encryption Syntax and Processing*, June 2001. <http://www.w3.org/TR/xmlenc-core/>.
29. World Wide Web Consortium (W3C), Note. *The Information Content Exchange Protocol*, October 1998. <http://www.w3.org/TR/NOTE-ice>.
30. World Wide Web Consortium (W3C), Working Draft. *SOAP Version 1.2*, July 2001. <http://www.w3.org/TR/soap12>.
31. XML Metadata Interchange (XMI) specification. <http://www.omg.org/cgi-bin/doc?ad/98-10-05>.
32. XML-RPC Home Page. <http://www.xmlrpc.com/>.

## A XML schemas

### A.1 Subject custom header block

```

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:t="http://www.w3.org/namespace/" >
  <xsd:element name='subject' >
    <xsd:complexType > <xsd:sequence>
      <xsd:element ref='t:user' /> <xsd:element ref='t:location' minOccurs='0' maxOccurs='1' />
      <xsd:element ref='t:role' maxOccurs='unbounded' />
    </xsd:sequence> </xsd:complexType>
  </xsd:element>
  <xsd:element name='user' >
    <xsd:complexType > <xsd:sequence>
      <xsd:element ref='t:userid' /> <xsd:element ref='t:passwdhash' />
    </xsd:sequence> </xsd:complexType>
  </xsd:element>
  <xsd:element name='location' >
    <xsd:complexType >
      <xsd:sequence>
        <xsd:element ref='t:symname' minOccurs='0' maxOccurs='1' />
        <xsd:element ref='t:netaddr' minOccurs='0' maxOccurs='1' />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name='role' >
    <xsd:complexType > <xsd:sequence>
      <xsd:element ref='t:roleid' /> <xsd:element ref='t:issuers' />
      <xsd:element ref='t:holder' /> <xsd:element ref='t:validity' />
    </xsd:sequence> </xsd:complexType>
  </xsd:element>
  <xsd:element name='issuer' >
    <xsd:complexType >
      <xsd:sequence>
        <xsd:choice>
          <xsd:element ref='t:public-key' /> <xsd:element ref='t:hash-of-key' />
          <xsd:element ref='t:name' />
        </xsd:choice>
        <xsd:element ref='t:turi' minOccurs='0' maxOccurs='unbounded' />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name='holder' >
    <xsd:complexType >
      <xsd:sequence>
        <xsd:choice>
          <xsd:element ref='t:public-key' /> <xsd:element ref='t:hash-of-key' />
          <xsd:element ref='t:name' />
        </xsd:choice>
        <xsd:element ref='t:turi' minOccurs='0' maxOccurs='unbounded' />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name='validity' >
    <xsd:complexType > <xsd:sequence>
      <xsd:element ref='t:notbefore' minOccurs='0' maxOccurs='1' />
      <xsd:element ref='t:notafter' minOccurs='0' maxOccurs='1' />
    </xsd:sequence> </xsd:complexType>
  </xsd:element>
</xsd:schema>

```

## A.2 Authorization syntax

```

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:t="http://www.w3.org/namespace/" >
  <xsd:element name='set_of_authorizations' >
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref='t:authorization' maxOccurs='unbounded' />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name='authorization' >
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref='t:subject' /> <xsd:element ref='t:object' /> <xsd:element ref='t:sign' />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name='subject' >
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref='t:id' /> <xsd:element ref='t:location' minOccurs='0' maxOccurs='1' />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name='id' >
    <xsd:complexType>
      <xsd:choice>
        <xsd:element ref='t:userid' /> <xsd:element ref='t:groupid' /> <xsd:element ref='t:roleid' />
      </xsd:choice>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name='location' >
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref='t:symname' minOccurs='0' maxOccurs='1' />
        <xsd:element ref='t:netaddr' minOccurs='0' maxOccurs='1' />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name='object' type='xsd:string' />
  <xsd:element name='sign' >
    <xsd:complexType>
      <xsd:attribute name='value' use='required' >
        <xsd:simpleType>
          <xsd:restriction base='xsd:string' >
            <xsd:enumeration value='+' />
            <xsd:enumeration value='- ' />
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name='userid' type='xsd:string' />
  <xsd:element name='groupid' type='xsd:string' />
  <xsd:element name='roleid' type='xsd:string' />
  <xsd:element name='symname' type='xsd:string' />
  <xsd:element name='netaddr' type='xsd:string' />
</xsd:schema>

```