CHAPTER 4

XML ACCESS CONTROL SYSTEMS: A COMPONENT-BASED APPROACH

E. Damiani[1] S. De Capitani di Vimercati[2] S. Paraboschi[3] P. Samarati[1]

(1) Università di Milano, Polo di Crema, 26013 Crema - Italy

(2) Università di Brescia, 25123 Brescia - Italy

(3) Politecnico di Milano, 20133 Milano - Italy

Abstract We recently proposed an access control model for XML information that permits the definition of authorizations at a fine granularity. We here describe the design and implementation of an *Access Control Processor* based on the above-mentioned model. We also present the major issues arising when integrating it into the framework of a component-based Web server system.

1. INTRODUCTION

XML [2] promises to have a great impact on the way information is exchanged between applications, going well beyond the original goal of being a replacement for HTML. Given the ubiquitous nature of XML, the protection of XML information will become a critical aspect of many security infrastructures. Thus, the investigation of techniques that can offer protection in a way adequate to the peculiarities of the XML data model is an important research goal.

Current solutions do not address the peculiarities of the security of XML information. Web servers may easily export XML documents, but their protection can typically be defined only at the file system level. Our proposal, presented in [3, 4], introduces an access control model for XML data that exploits the characteristics of XML documents, allowing the definition of access control policies that operate with a fine granularity, permitting the definition of authorizations at the level of the single element/attribute of an XML document.

The focus of this paper is the design and implementation of a system offering the services of our access control model. We first give in Section 2 a brief description of the approach. Then, in Section 3 we describe the high-level software architecture. Section 4 presents the IDL interfaces of the classes which implement the services of the access control system. Finally, Section 5 is dedicated to the integration of the access control system with Web based systems.

The analysis contained in this paper derives from the experience we gained in the implementation of the current prototype of the system; our results should be helpful to those considering the implementation of security mechanisms in the WWW/XML context.

2. XML ACCESS CONTROL MODEL

The access control model we present is based on the definition of authorizations at the level of the elements and attributes of an XML document.

A natural interpretation for XML documents is to consider them as trees, where elements and attributes correspond to nodes, and the containment relation between nodes is represented by the tree arcs. Authorizations can be *local*, if the access privilege they represent applies only to a specific element node and its attributes, or can be *recursive*, if the access is granted/denied to the node and all the nodes descending from it (i.e., the nodes that in the textual representation of an XML document are enclosed between the start and end tags).

We identified two levels at which authorizations on XML documents can be defined, instance and DTD (Document Type Definition, a syntax defining the structure of the document). DTD level authorizations specify the privileges of all the documents following a given DTD, whereas instance level authorizations denote privileges that apply only to a specific document. The distinction between the two authorization types may correspond to the distribution of responsibilities in an organization, as DTD authorizations may be considered derived from the requirements of the global enterprise, whereas authorizations on the instance may be the responsibility of the creator of the document. We also hypothesize that normal DTD authorizations are dominated by instance level ones (following the general principle that more specific authorizations win [6, 9] and that an instance level authorization is more specific than a DTD level one), but we also consider the need for an organization to have assurance that some of the DTD authorizations are not overruled. Thus, we permit the definition of *hard* DTD authorizations, which dominate instance level ones. For cases where instance level authorizations must be explicitly defined as valid only if not in conflict with DTD level ones, we designed *soft* instance level authorizations.

Each authorization has five components: *subject*, *object*, *type*, *action* and *sign*. The subject is composed by a triple that describes the user or the group of users to which the authorization applies, combined with the numeric (IP) and symbolic (DNS) addresses of the machine originating the request. This triple can thus permit to define controls that consider both the user and the location. Wild card character `*` permits the definition of patterns for addresses (e.g., `131.*` for all IP addresses having `131` as first component, or `*.it` for all addresses in the Italian domain). The authorization applies on the request only if the triple of parameters of the requester is equal or more specific in all three components of the authorization subject. For example, an authorization with subject `<Student,131.175.*,*.polimi.it>` will be applied to a request from <`Ennio,131.175.16.43,pcenn.elet.polimi.it`>, if `Ennio` is a member of group `Student`. The object is identified by means of an XPath [13] expression. XPath expressions may be used to identify document components in a declarative way, but they can also use navigation functions, like `child`, offering a standard and powerful way to identify the elements and attributes of an XML document. The type can be one of eight values, arising from the combination of three binary properties: DTD level or instance level; local or recursive; normal or soft/hard. The eight types, in order of priority, are: local DTD level hard (LDH), recursive DTD level hard

(RDH), local instance level (L), recursive instance level (R), local DTD level (LD), recursive DTD level (RD), local instance level soft (LS), recursive instance level soft (RS). Since currently, most XML applications offer read-only access, the action currently supported by our prototype is only *read*.
A positive authorization sign specifies that the authorization permits access, a negative sign instead forbids it.

Authorizations are then evaluated according to the following principles:

- If two authorizations are of a different type, the one with the higher priority wins (e.g., between LD and LS, LD wins).
- If two authorizations have the same type, but the object of one is more specific, the more specific wins (e.g., a recursive authorization for an element is dominated by authorizations on its subelements).
- If two authorizations have the same type and are on the same object, but the subject of one is more specific, the more specific wins (e.g., an authorization for the `Public` group is dominated by an authorization for the specific user `Ennio`).
- When none of the above criteria is met, a site-specific general resolution policy is used (e.g., assuming a closed access control policy, the negative authorization wins).

We refer to the presentations in [3, 4] for a complete overview of the characteristics of our solution. In this paper we intend to focus on the design and implementation of a system for access control.

3. SOFTWARE ARCHITECTURE: AN OUTLINE

For the access control technique outlined in Section 2 to be of any interest from the software designer point of view, it must be suitable for clean integration in the framework of XML-based WWW applications. To clarify this point, we shall briefly introduce the use of an *XML Access Control Processor* (ACP) as a part of a *component-based Web service* [5], where a set of reusable components are responsible of processing user *requests*.

The sample *UML Sequence Diagram* shown in Figure 1 gives a general idea of the internal operation of our processor and of its integration in a Web server system. For the sake of simplicity, in this Section we shall not deal with the transformation of the XML document, which is hidden inside a container ACP object. Also, Figure 1 does not show provisions for persistence management and caching. The ACP object wraps up entirely the computation of access permissions to individual elements and the final transformation to be performed on the XML document. The standard operation of a Web server receiving a HTTP request (**1**) from a user is represented in Figure 1 by the creation of a transient *Connection Handler* object (**2**). Then, a *Processor* is activated by the Connection Handler, and an *ACP object* is instantiated (**3**). In turn, ACP creates a *Subjects* object which fully encapsulates the subjects' hierarchy (**4**). After getting the available data about the user/group of the requestor, together with the IP address and symbolic name (**5**), ACP signals to a static *Loader/Parser* object to upload the requested XML document (**6**). The Loader/Parser translates the document into a low level object data structure based on the *Document Object model (DOM)* (not shown in Figure 1) more suitable for modification. Then, the ACP

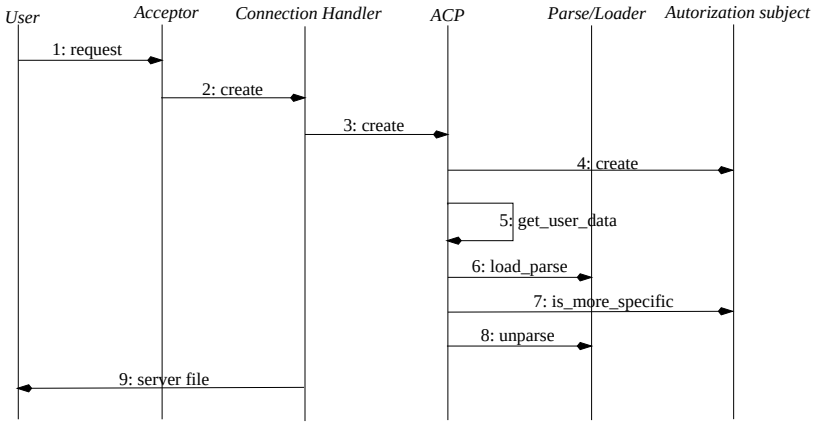

Figure 1 Sequence diagram

modifies the data structure according to the permissions, using the services of the transient *Subjects* object which fully encapsulates the subjects' hierarchy. Messages sent to the *Subjects* object (**7**) allow the ACP object to position the requestor in the subjects' hierarchy. After computing the transformation, the ACP object signals to the Parser (**8**) that the data structure can be returned to its text format, ready to be served to the user by the Connection Handler (**9**).

From the architectural point of view, it should be noted that our design is fully *server side*: all the message exchanges of Figure 1 except the connection itself (**1**) take place on the server. *Client-side* processing strategies (including client-side caching, and caching proxy servers) have been traditionally used for HTML. However, client-side solutions have been found to be less apt at XML-based Web services [5], where the contents to be transferred usually require extra processing. There may well be cases where *negotiation* could be envisioned between the client and the server as to the kinds of XML content transformations that are possible by the server and acceptable to the client; but it is clear that client-side techniques must be excluded from any sound implementation of access control. As we will see in Section 4.3, the fictitious ACP object is indeed a complex object inheriting from Java *Servlet* class.

4. THE XML-AC PACKAGE

The interface offered by the XML-AC system can be represented by a set of classes modeling the entities and concepts introduced by the access control model. Two major class families are used: one constitutes an extension of the DOM Interface defined by the W3C, the other describes all the concepts on which the ACP system is based.

4.1. ARCHITECTURAL OBJECTS: THE SECUREDOM HIERARCHY

Our system, like most XML applications, internally represents XML documents and DTDs as object trees, according to the Document Object Model (DOM) specification [12]. DOM provides an object-oriented *Application Program Interface* (API) for HTML and XML documents. Namely, DOM defines a set of object definitions (e.g.,

`Element`, `Attr`, and `Text`) to build an object-oriented representation which closely models the document structure. While DOM trees are topologically equivalent to XML trees, they represent element containment by means of the object-oriented *part-of* relationship. For example, a document element is represented in DOM by an `Element` object, an element contained within another element is represented as a child `Element` object, and text contained in an element is represented as a child `Text` object. The root class of the DOM hierarchy is `Node`, which represents the generic component of an XML document and provides basic methods for insertion, deletion and editing; via inheritance, such methods are also defined for more specialized classes in the hierarchy, like `Element`, `Attr` and `Text`. `Node` also provides a powerful set of navigation methods, such as `parentNode`, `firstChild` and `nextSibling`. Navigation methods allow application programs to visit the DOM representation of XML documents via a sequence of calls to the interface. Specifically, the `NodeList` method, which returns an array containing all the children of the current node, is often used to explore the structure of an XML document from the root to the leaves.

We extended the DOM hierarchy associating to the members of the class hierarchy a `Secure` variant. Each `Secure` variant extends the base class with references to all the authorizations which can be applied to the node. Internally, each class separates the references to authorizations into 8 containers, depending on the authorization type. Each container internally keeps a list of positive and negative authorizations of the type. The IDL interface common to all *Secure* classes, written in IDL, the OMG-CORBA standard *Interface Definition Language*, is:

```
interface Secure{
    void addAuthorization (in Authorization AuthToAdd);
    AuthorizationLabel defineFinalLabel
                      (in AuthorizationLabel FatherAuthRecHard,
                       in AuthorizationLabel FatherAuthRec,
                       in AuthorizationLabel FatherAuthRecDTD,
                       in AuthorizationLabel FatherAuthRecSoft);
    void prune();}
```

From this interface is possible to define the interfaces of each `Secure` variant of the DOM classes, using multiple inheritance in IDL definitions. For example, the definition of the `SecureNode` class is `interface SecureNode: Node, Secure {}`.

The extension imposes a limited increase in the cost of the document representation. Indeed, the containers can be implemented with dynamic structures, occupying space only when authorizations are actually associated with the node. The node contains references to the full description of the authorizations, kept in a separate area of memory. In this way, there are no redundancies and, since in the evaluation of access control authorizations must not be modified, the use of references is fully adequate.

4.2. APPLICATION OBJECTS: THE ACCESS CONTROL CLASSES

We describe here the main classes of the Access Control Processor: `UserGroup`, `User`, `AuthorizationLabel`, `AuthorizationType`, `AuthorizationSubject` and finally `Authorization`.

Class `UserGroup` describes the features common to a user and a group: both have a name and appear in the user/group hierarchy. The services offered by the class are the storage of the hierarchy on users/groups, method `addDescendent` that permits to add a new user/group in the hierarchy, and method `isEqualOrMoreSpecific` that permits to determine if a user/group belongs, directly or indirectly, to another user/group.

```
interface UserGroup{
    attribute string Name;
    void addChild (in UserGroup ChildToAdd);
    boolean isEqualOrMoreSpecific (in UserGroup UserGroupToCompare);}
```

Class `User` is a specialization of class `UserGroup` and extends it with all the information specific to users, like the real person name. Method `checkPassword` implements the cryptographic function that determines if the password returned by the user corresponds to the stored value.

```
interface User:  UserGroup{
    attribute string FirstName;
    attribute string LastName;
    boolean checkPassword(in string PasswordToCheck);
    void setPassword(in string NewPassword);}
```

Class `AuthorizationLabel` contains an enumerative type that describes the three values (positive, negative, and undefined) of the security label that can be assigned to a node, after the evaluation of the existing authorizations. Its methods permit to set and retrieve the value.

```
interface AuthorizationLabel{
    enum Label_t (positive, negative, undefined);
    attribute Label_t label;
    void setPositive();
    void setNegative();
    void setUndefined();
    boolean isPositive();
    boolean isNegative();
    boolean isUndefined();}
```

Class `AuthorizationType` describes the possible types of authorization. Its methods permit to set and to retrieve the authorization type (local or recursive, on the document or on the DTD, and hard or soft).

```
interface AuthorizationType{
    enum AuthType_t (LDH, RDH, L, R, LD, RD, LS, RS);
    void setLocal();
    void setRecursive();
    void setOnInstance();
    void setOnInstanceSoft();
    void setOnDTD(); }
    void setOnDTDHard(); }
    boolean isLocal();
    boolean isRecursive();
    boolean isOnInstance();
    boolean isOnInstanceSoft();
    boolean isOnDTD();
    boolean isOnDTDHard(); }
```

Class `AuthorizationSubject` describes the triple ⟨user-group, IP address, symbolic address⟩ that identifies the subjects to which the authorizations must be applied. The class offers methods to get and assign the components of the addresses and a method `isEqualOrMoreSpecific` to determine if one subject is equal or more specific than another subject.

```
interface AuthorizationSubject{
    void setUserGroup(in UserGroup userGroupToSet);
    UserGroup getAuthUser();
    void setIpAddress(in string IPAddrToSet);
```

```
    string getIpAddress();
    void setSnAddress(in string SymbAddrToSet);
    string getSnAddress();
    boolean isEqualOrMoreSpecific(in AuthorizationSubject AuthSubjToCmp);}
```

Class `Authorization` represents the authorizations that are defined on the system. Each authorization is characterized by a subject (class `AuthorizationSubject`), an object (represented by an XPath expression, managed by classes defined in an external XSL implementation), the sign (represented by an `AuthorizationLabel` component for which value *undefined* is not admitted), the action (currently a simple string), and finally the type (represented by a component of class `AuthorizationType`).

```
interface Authorization{
    attribute AuthorizationSubject subject;
    attribute XPathExpr object;
    attribute AuthorizationLabel sign;
    attribute AuthorizationType type;
    attribute string action; }
```

4.3. DEPLOYING THE PACKAGE

We implemented the above classes in Java and used them to realize a prototype of the Access Control Processor with a Java servlet solution. Java servlets, designed by Sun and part of the Java environment, appear as a set of predefined classes that offer services that are needed for the exchange of information between a Web server and a Java application. Examples of these classes are `HttpSession` and `HttpRequest`. Java servlets constitute a simple and efficient mechanism for the extension of the services of a generic Web server; the Web server must be configured to launch the execution of a Java Virtual Machine when a request for a URL served by a servlet arrives, passing the parameters of the request with a specified internal protocol.

The Java classes we implemented can also be used in a different framework, using a solution like JSP (Java Server Pages). Actually, JSP is internally based on servlets, but it offers an easier interface to the programmer, requiring the definition of HTML/XML templates which embed the invocation of servlet services. We have already demonstrated the use of the prototype inside a JSP server.

There are several other architectures that could be used and whose applicability we plan to investigate in the future. Since we gave an IDL description of the classes that constitute the implementation of our system, it is natural to envision a solution based on the distributed object paradigm, using protocols like RMI/IIOP (for the Java implementation) or the services of a generic CORBA broker (where the services are implemented by objects written in a generic programming language).

5. INTEGRATION WITH WEB-BASED SYSTEMS

We are now ready to describe how our access control system can be integrated in a Web-based framework for distribution and management of XML information. This architecture needs to include a number of components and a careful study of their interaction with access control is of paramount importance to achieve an efficient implementation.

5.1. LINKING XAS TO XML DOCUMENTS AND DTDS

As XASs contain access control information for XML documents and DTDs, links must be provided allowing the system, upon receipt of a HTTP request for an XML document, to locate the XAS associated with both the document itself and its DTD. In current XML practice, association between XML documents and their DTDs is made by either *direct inclusion* (the DTD is embedded in the XML document) or by *hypertext link* (the XML document contains the URL of its DTD). Neither technique seems appropriate for linking documents and DTDs to XASs as they would interfere with the normal processing of XML documents, and pose the problem of managing access control for legacy documents not linked to any XAS specification. Luckily enough, we can rely on the abstract nature of XML *XLink* specification to define *out-of-line* links that reside *outside* the documents they connect, making links themselves a viable and manageable resource. The repertoire of out-of-line links defining access control mappings is itself an XML document, easily managed and updated by the system manager; nonetheless it is easily secured by standard file-system level access control. We propose to set up a suitable *namespace*, called `AC`, which is for the time being aimed at reserving the standard tag name `<XAS>` to denote off-line links between documents, DTDs and XASs. The DTD of the documents containing the mappings from XML documents to DTDs and to XASs can be written as follows:

```
<!ENTITY % xlink " type    CDATA  # FIXED 'arc'
                   role    CDATA  'access control'
                   title   CDATA  'access control'
                   actuate CDATA  # FIXED 'auto'
                   from    CDATA  # REQUIRED
                   to      CDATA  # REQUIRED">

<!ELEMENT XAS EMPTY>
<!ATTLIST XAS  % xlink
         xmlns:xlink CDATA  'http://www.w3.org/TR/xlink' >
```

Note that, in the private documents specifying link sets for each site and at the DTD level, the name of the XAS element will be preceded by the mention of the `AC` namespace in order to avoid ambiguity. In the above DTD definition, we rely on a reusable XML *entity* to group the attributes needed to set up an out-of-line link between a document and its access control information. Namely, out-of-line links are identified by the `type` attribute being set to `"arc"`, and by the presence of required `from` and `to` attributes instead of the usual `href` used for embedded links. The `actuate` attribute is set to `"auto"`, meaning that the traversal of the link will be automatically made by the system and not revealed to the user. Finally, the `role` and `title` attributes are used primarily for descriptive purposes and are therefore not mandatory.

5.2. XML-AC SUPPORT FOR SESSIONS

In the current prototype, sessions are managed by class `HttpSession`, a component of the Java servlet environment. Class `HttpSession` keeps track of the series of requests originating from the same user. Using the services of `HttpSession` it is possible to ask only once to the user to declare his identity and password. The implementation of class `HttpSession` permits to manage sessions in two modes, with or without cookies. When the client has cookies enabled, `HttpSession` may store a

session identifier in the client cookies and use it to identify the request; if cookies are not enabled, sessions are identified by storing the session identifier as a parameter of the requests that are embedded into the page which is returned to the user. Since users often do not enable cookies, it is important to be able to manage sessions independently.

We observe that the solution we implemented, based on the services of class `HttpSession`, is adequate for our context, where the goal was a demonstration of the capabilities of the access control model. An environment with strong security requirements should probably plan a different implementation of the session management services, using adequate cryptographic techniques to protect the connection.

5.3. A MULTITHREADED SERVER FRAMEWORK

To guarantee efficient and effective integration of access-control in the framework of Web-based systems, two basic problems must be solved:

Quality of Service The emergence of the World Wide Web as a mainstream technology has highlighted the problem of providing a high quality of service (*QoS*) to application users. This factor alone cautioned us about the risk of increasing substantially the processing load of Web server.

Seamless Integration A second point to be mentioned regards how to provide XML access control as seamlessly as possible, without interfering with the operation of other presentation or data-processing services. Moreover, the access control service should be introduced on existing servers with minimal or no interruption of their operation.

To deal with these problems, we chose an integrated (yet modular) approach, that supports reuse allowing for different deployment solutions according to implementation platforms' performance profiles. In fact, besides being deployed as a single-thread servlet invoked by the Connection Handler, as in our current prototype, our processor can be easily interfaced to a *Dispatcher* registered with an *Event Handler*. Dispatcher-based multi-threading can be managed *synchronously*, according to the well known *Reactor/Proactor* design pattern [7] or *asynchronously*, as in the *Active Object* pattern. In this section we shall focus on the former choice, as it facilitates integration of our XML access control code in the framework of existing general-purpose server-side transformers based on the same design pattern like *Cocoon* [1]. Figure 2 depicts the Reactor-based multi-threading technique.

In order to avoid being a potential bottleneck for the server operation, our Access Control system needs to manage effectively a high number of *concurrent requests*. Multi-threaded designs are currently the preferred choice to implement Web-based, high-concurrency systems. This is also our design choice for our components. However, it must be noted that no Java-based design of multi-threading components has full control on thread management: when running on an operating system that supports threads, the *Java Virtual Machine* automatically maps Java threads to native threads [8], while when no native thread support is available, the JVM has to emulate threads. In the latter case, the emulation technique chosen by the JVM implementors can make significant difference in performance. In the sequel, we shall briefly describe the Java thread management technique used for the implementation of our processor, providing full synchronization between threads when accessing the same DOM and `AuthorizationSubject` objects. To clarify the synchronization problem associated with multi-threading, consider two access control tasks that need to be executed in

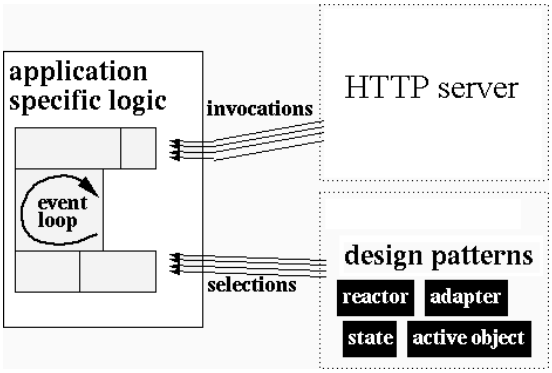

Figure 2 Cocoon-style multi-threading technique

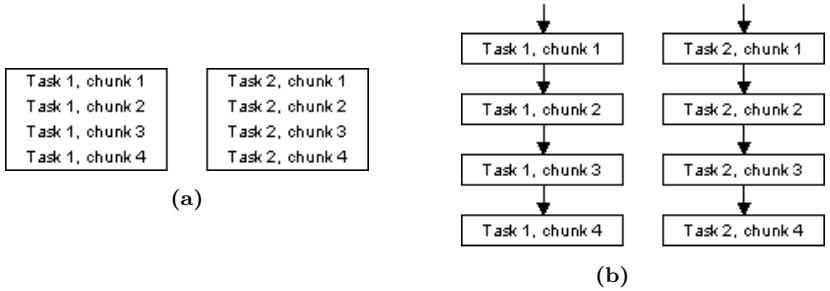

Figure 3 Two AC tasks to be executed in parallel (a) and their subdivision into four atomic sub-tasks (b)

parallel (see Figure 3(a)). For the sake of simplicity both tasks are naturally subdivided into four *atomic* non-interruptible sub-tasks, loosely corresponding to actions from **(4)** to **(7)** of Section 3. In a "naive" multi-threaded implementation of our processor, each task would be executed on its own thread. However, the only way to preserve atomicity using this technique would be to explicitly synchronize threads by means of *semaphores*. Fortunately, the additional complexity and overhead involved in explicit synchronization can be easily avoided in our case.

Synchronous dispatching A synchronous dispatcher can be used to solve the synchronization problem by simulating multi-threading within a single Java thread. To illustrate the evolution of our design from a single task divided into portions to a synchronous dispatcher, consider first the subdivision of each task of Figure 3(a) into four independent sub-tasks, depicted in Figure 3(b). From the Java implementation point of view, each sub-task can now be straightforwardly defined as the `run()` method of a `Runnable` object [10]. Then, the objects can be stored into an array, and a *scheduler* module can be added executing the objects one at a time. `Sleep()` or `yield()` calls mark the transition between sub-tasks.

As anticipated, this code is a simple implementation of Schmidt's Reactor design pattern [7]. The effect is essentially the same as several threads waiting on a single *ordered binary semaphore* that is set to `true` by an event. Here, the programmer

```
Runnable[] task = new Runnable[]
{
 new Runnable(){ public void run(){ /* execute sub-task 1 */ } },
 new Runnable(){ public void run(){ /* execute sub-task 2 */ } },
 new Runnable(){ public void run(){ /* execute sub-task 3 */ } },
 new Runnable(){ public void run(){ /* execute sub-task 4 */ } },
};
for( int i = 0; i < task.length; i++ )
{task[i].run();
 Thread.getCurrentThread().yield();
}
```

Figure 4 Sample Java code for the synchronous dispatcher

```
Runnable[] two_tasks = new Runnable[]
{
 new Runnable(){ public void run(){ /* execute task 1, sub-task 1 */ } },
 new Runnable(){ public void run(){ /* execute task 2, sub-task 1 */ } },
 new Runnable(){ public void run(){ /* execute task 1, sub-task 2 */ } },
 new Runnable(){ public void run(){ /* execute task 2, sub-task 2 */ } },
 new Runnable(){ public void run(){ /* execute task 1, sub-task 3 */ } },
 new Runnable(){ public void run(){ /* execute task 2, sub-task 3 */ } },
 new Runnable(){ public void run(){ /* execute task 1, sub-task 4 */ } },
 new Runnable(){ public void run(){ /* execute task 2, sub-task 4 */ } },
};
for( int i = 0; i < two_task.length; i++ )
 { two_tasks[i].run();
 Thread.getCurrentThread().yield();
}
```

Figure 5 The interleaving dispatcher

retains full control over the sequence of subtask execution after the event. In our AC processor, however, a slightly more complex technique should be used, as we need to execute complex transformation tasks concurrently, each of them being subdivided into atomic sub-tasks. To deal with this problem, the synchronous dispatcher of Figure 4 can be easily modified [10] to provide *interleaving* (Figure 5).

The behavior of the code in Figure 5 allows for a multi-threading *cooperative* system (in which threads explicitly yield control to other threads). Of course, this synchronous dispatching technique is aimed at native multi-threaded operating systems, where all the subtasks are executing on a single operating system-level thread. In this case, there is no synchronization overhead at all, and no expensive context switch into the host operating system's kernel. It should be noted that several dispatchers could be used, each running on its own thread (as in Sun's *green thread* model [11]), so that cooperative and preemptive threads may share the same process.

6. CONCLUSION

In this paper we presented the major results of the study we did before the implementation of the processor for the proposed access control model for XML data. Most of the considerations we present are not specific to our system, but can be of interest in any context where services for the security of XML must be implemented.

There are several directions where our work can be extended and that offer interesting opportunities. For instance, we focused on multi-threading techniques to obtain efficient concurrent execution of access control tasks. However, synchronization overhead is obviously not the only performance problem. Other techniques rather than round-robin interleaving could be adopted: e.g., the XML access-control service could adaptively optimize itself to provide higher priorities for smaller requests. These techniques combined could potentially produce a system highly responsive and with an adequate throughput. The next release of the ACP plans to implement the prioritized strategy.

Acknowledgments

The authors wish to thank Daniel Menasce' for interesting discussions about XML processing performance issues.

References

[1] Apache Software Foundation. Cocoon, a Java publishing framework. http://xml.apache.org/cocoon, 2000.

[2] T. Bray et.al. (ed.). *Extensible Markup Language (XML) 1.0*. World Wide Web Consortium (W3C), February 1998. http://www.w3.org/TR/REC-xml.

[3] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. Design and implementation of an access control processor for XML documents. In *Proc. of the Ninth Int. Conference on the World Wide Web*, Amsterdam, May 2000.

[4] E. Damiani, S. De Capitani Di Vimercati, S. Paraboschi, and P. Samarati. Securing XML documents. In *Proc. of EDBT 2000*, Konstanz, Germany, March 2000.

[5] J. Hu, I. Pyarale, and D. Schmidt. Applying the proactor pattern to high performance web services. In *Proc. of the 10th International Conference on Parallel and Distributed Computing*, Las Vegas, Nevada, October 1998.

[6] S. Jajodia, P. Samarati, and V. S. Subrahmanian. A Logical Language for Expressing Authorizations. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 31–42, Oakland, CA, May 1997.

[7] R. G. Lavender and D. Schmidt. Reactor: A object behavioral pattern for concurrent programming. In J. Vlissides, D. Coplien, and M. Kerth, editors, *Pattern Languages of Program Design 2*. Addison Wesley, 1995.

[8] D. Lea. *Concurrent Programming in Java*. Addison Wesley, 1996.

[9] T.F. Lunt. Access Control Policies for Database Systems. In C.E. Landwehr, editor, *Database Security, II: Status and Prospects*, pages 41–52. North-Holland, Amsterdam, 1989.

[10] B. Marchant. Multithreading in java. http://www.javacats.com/US/articles/multithreading.html, 1996.

[11] M. L. Powell, S. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks. *SunOS Multi-thread Architecture*. Sun Microsystems, 1998.

[12] World Wide Web Consortium (W3C). *Document Object Model (DOM) Level 1 Specification Version 1.0*, October 1998. http://www.w3.org/TR/REC-DOM-Level-1.

[13] World Wide Web Consortium (W3C). *XML Path Language (XPath)*, November 1999. http://www.w3.org/TR/xpath.