Distributed and Cooperative Updates of XML Documents

A Ph.D Thesis in Computer Science presented
by

Giovanni Mella

Internal Advisors: Prof. E. Bertino and Prof. E. Ferrari
External Advisors: Prof. N. Li and Prof. J. Joshi

Università degli Studi di Milano
Dipartimento di Informatica e Comunicazione
November 2004
# Contents

Title Page ........................................ i  
Table of Contents ...................................... ii  
List of Figures ........................................ vi  
List of Tables ........................................ viii  
Dedication ........................................ ix  

## 1 Introduction 1

## 2 Related work 6

2.1 Introduction ...................................... 7
2.2 Access Control Models and Mechanisms ....................... 7
2.3 XML Document Updates ................................ 10
2.4 Encryption and Digital Signature Techniques .................. 11
2.5 Delegation of Authorizations ................................ 12
2.6 Collaborative Computing .................................. 14
2.7 Group Communication ................................... 15

## 3 A Framework for distributed and cooperative updates of XML Documents 17

3.1 Introduction ...................................... 18
3.2 Basic Concepts of XML ................................. 20
3.3 An Access Control Model for XML Documents .................. 21
3.4 Document Encryption ................................... 25
3.5 System Implementation .................................. 29

## 4 Cooperative updates in non-colluding byzantine distributed systems 31

4.1 Introduction ...................................... 33
4.2 Assumptions ....................................... 34
4.3 Generation of Control Information .......................... 35
4.3.1 Authoring Certificates .............................. 36
4.3.2 Control data structures for document regions ............... 37
4.3.3 Generalized control information ......................... 42
4.4 Subject and Document Originator Protocols .................... 44
4.4.1 Subject Protocol ...................................... 44
4.4.1.1 Package integrity and authentication ................. 45
## Contents

<table>
<thead>
<tr>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4.1.2 Package update and delivery</td>
</tr>
<tr>
<td>4.4.1.3 An Illustrative Example</td>
</tr>
<tr>
<td>4.4.1.4 Correctness of the Subject Protocol</td>
</tr>
<tr>
<td>4.4.2 Document Originator Protocol</td>
</tr>
<tr>
<td>4.5 Complexity Analysis of the Proposed Approach</td>
</tr>
<tr>
<td>4.5.1 Communication cost</td>
</tr>
<tr>
<td>4.5.2 Size of exchanged information</td>
</tr>
<tr>
<td>4.5.3 Number of certificates generated for a document $d$</td>
</tr>
<tr>
<td>4.5.4 Execution time of the integrity check protocol and distributed view generation/decryption vs execution time of the centralized view generation/encryption and distributed decryption</td>
</tr>
</tbody>
</table>

5 Cooperative updates in colluding byzantine distributed systems | 67 |
| 5.1 Introduction | 68 |
| 5.2 Preliminary definitions | 70 |
| 5.3 Control information | 71 |
| 5.4 Protocol for managing distributed updates | 76 |
| 5.4.1 Path and deltas integrity check | 78 |
| 5.4.2 Document integrity check | 80 |
| 5.4.3 Protocol for document and path modification | 81 |
| 5.5 A running example | 84 |
| 5.6 Correctness proof | 85 |

6 Cooperative updates in colluding byzantine and failure prone distributed systems | 88 |
| 6.1 Introduction | 91 |
| 6.2 Overview | 92 |
| 6.2.1 Document Originator | 93 |
| 6.2.2 Cooperative Group | 93 |
| 6.2.3 Delegates Group | 94 |
| 6.3 Specification Languages | 94 |
| 6.3.1 Flow Policy Specification Language | 94 |
| 6.3.2 Flow Modification Specification Language | 96 |
| 6.4 Control Information | 100 |
| 6.4.1 Preliminary Concepts | 101 |
| 6.4.2 Document Control Information | 106 |
| 6.4.3 Modification declarations | 109 |
| 6.5 Distributed and cooperative update process | 111 |
| 6.5.1 Assumptions and Properties | 112 |
| 6.5.2 Protocol Parameters Setting | 115 |
| 6.5.3 Update process phases | 117 |
| 6.5.3.1 Initialization phase | 117 |
| 6.5.3.2 Cooperative update phase | 117 |
| 6.5.3.3 Final phase | 129 |
| 6.5.4 Formal results | 129 |
| 6.5.5 Recovery | 130 |
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Overview of the update approach</td>
<td>19</td>
</tr>
<tr>
<td>3.2</td>
<td>(a) An example of XML document and (b) its graph representation</td>
<td>20</td>
</tr>
<tr>
<td>3.3</td>
<td>Examples of ( \lambda )-sec credentials</td>
<td>22</td>
</tr>
<tr>
<td>3.4</td>
<td>An example of Policy Base</td>
<td>26</td>
</tr>
<tr>
<td>3.5</td>
<td>The ( \lambda )-Sec Policy Base template</td>
<td>27</td>
</tr>
<tr>
<td>3.6</td>
<td>Author-( \lambda ) architecture</td>
<td>29</td>
</tr>
<tr>
<td>4.1</td>
<td>Distributed document updates: overall schema</td>
<td>34</td>
</tr>
<tr>
<td>4.2</td>
<td>Modification of the structure ( MR_d ) along an hypothetical document-path</td>
<td>43</td>
</tr>
<tr>
<td>4.3</td>
<td>Graphic representation of a package</td>
<td>44</td>
</tr>
<tr>
<td>4.4</td>
<td>An algorithm for verifying package integrity and authenticity</td>
<td>45</td>
</tr>
<tr>
<td>4.5</td>
<td>Main steps of the package integrity and authenticity algorithm</td>
<td>46</td>
</tr>
<tr>
<td>4.6</td>
<td>Update and delivery processes</td>
<td>48</td>
</tr>
<tr>
<td>4.7</td>
<td>Update flow of a modifiable region (Step 1)</td>
<td>50</td>
</tr>
<tr>
<td>4.8</td>
<td>Update flow of a modifiable region (Step 2)</td>
<td>51</td>
</tr>
<tr>
<td>4.9</td>
<td>Update flow of a modifiable region (Step 3)</td>
<td>52</td>
</tr>
<tr>
<td>5.1</td>
<td>Document pre-processing</td>
<td>69</td>
</tr>
<tr>
<td>5.2</td>
<td>An example of document path</td>
<td>72</td>
</tr>
<tr>
<td>5.3</td>
<td>Document package structure</td>
<td>74</td>
</tr>
<tr>
<td>5.4</td>
<td>Example of data structures within a document package</td>
<td>75</td>
</tr>
<tr>
<td>5.5</td>
<td>Example of document path extension</td>
<td>82</td>
</tr>
<tr>
<td>5.6</td>
<td>Effective path checklist data structures, when the security level is set to 2</td>
<td>83</td>
</tr>
<tr>
<td>5.7</td>
<td>Example of collusion detection using the effective path checklists</td>
<td>84</td>
</tr>
<tr>
<td>6.1</td>
<td>(a) An example of XML flow policy and (b) its graph representation</td>
<td>97</td>
</tr>
<tr>
<td>6.2</td>
<td>An example of Rule Base</td>
<td>100</td>
</tr>
<tr>
<td>6.3</td>
<td>Message exchange</td>
<td>116</td>
</tr>
<tr>
<td>6.4</td>
<td>Sub-phases messages exchange</td>
<td>121</td>
</tr>
<tr>
<td>6.5</td>
<td>Document Originator Protocol</td>
<td>122</td>
</tr>
<tr>
<td>6.6</td>
<td>Subject Protocol</td>
<td>123</td>
</tr>
<tr>
<td>6.7</td>
<td>Delegate Protocol</td>
<td>125</td>
</tr>
<tr>
<td>6.8</td>
<td>Down Delegate Protocol</td>
<td>126</td>
</tr>
<tr>
<td>6.9</td>
<td>The recovery algorithm (part 1)</td>
<td>134</td>
</tr>
</tbody>
</table>
List of Figures

6.10 The recovery algorithm (part 2) ........................................ 135
6.11 An Algorithm for the merging of recovery responses ............... 138
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Access privileges and their semantics</td>
<td>23</td>
</tr>
<tr>
<td>3.2</td>
<td>Table Key-info for the document in Figure 3.2</td>
<td>28</td>
</tr>
<tr>
<td>4.1</td>
<td>Control data structures for non-modifiable regions</td>
<td>37</td>
</tr>
<tr>
<td>4.2</td>
<td>Components of the control data structures for non-modifiable regions</td>
<td>38</td>
</tr>
<tr>
<td>4.3</td>
<td>Control data structures for modifiable regions</td>
<td>39</td>
</tr>
<tr>
<td>4.4</td>
<td>Components of the control data structures for modifiable regions</td>
<td>40</td>
</tr>
<tr>
<td>4.5</td>
<td>Parameters involved in the complexity/cost analysis</td>
<td>58</td>
</tr>
<tr>
<td>4.6</td>
<td>Size of the building blocks of a package/acces request</td>
<td>61</td>
</tr>
<tr>
<td>4.7</td>
<td>Size of the basic package structures</td>
<td>62</td>
</tr>
<tr>
<td>4.8</td>
<td>Time required to perform the basic integrity check protocol operations</td>
<td>65</td>
</tr>
<tr>
<td>4.9</td>
<td>Time required to perform the integrity check protocol operations</td>
<td>65</td>
</tr>
<tr>
<td>6.1</td>
<td>Structure of a flow policy</td>
<td>95</td>
</tr>
<tr>
<td>6.2</td>
<td>Components of the structure of a flow policy</td>
<td>96</td>
</tr>
<tr>
<td>6.3</td>
<td>Possible configurations of a flow modification rule and their semantics</td>
<td>99</td>
</tr>
<tr>
<td>6.4</td>
<td>Table containing flow policy regions for the flow policy in Figure 6.1</td>
<td>102</td>
</tr>
<tr>
<td>6.5</td>
<td>Control data structures for deletable and non-deletable document atomic elements</td>
<td>105</td>
</tr>
<tr>
<td>6.6</td>
<td>Components of the control data structures for deletable and non-deletable document atomic elements</td>
<td>106</td>
</tr>
<tr>
<td>6.7</td>
<td>Control data structures for non-modifiable document regions</td>
<td>107</td>
</tr>
<tr>
<td>6.8</td>
<td>Components of the control data structures for non-modifiable document regions</td>
<td>107</td>
</tr>
<tr>
<td>6.9</td>
<td>Modifiable Region Classification</td>
<td>108</td>
</tr>
<tr>
<td>6.10</td>
<td>Control data structures for modifiable document regions</td>
<td>108</td>
</tr>
<tr>
<td>6.11</td>
<td>Components of the control data structures for FDUR</td>
<td>109</td>
</tr>
<tr>
<td>6.12</td>
<td>Modification declaration structure within a receiver specification</td>
<td>110</td>
</tr>
<tr>
<td>6.13</td>
<td>Components of the Modification Declaration Structure</td>
<td>111</td>
</tr>
<tr>
<td>6.14</td>
<td>Structure of the Invalid Modification Document Declaration information</td>
<td>113</td>
</tr>
<tr>
<td>6.15</td>
<td>Data structures used by the protocols</td>
<td>114</td>
</tr>
<tr>
<td>6.16</td>
<td>Messages (part 1)</td>
<td>118</td>
</tr>
<tr>
<td>6.17</td>
<td>Messages (part 2)</td>
<td>119</td>
</tr>
<tr>
<td>6.18</td>
<td>Functions (part 1)</td>
<td>127</td>
</tr>
<tr>
<td>6.19</td>
<td>Functions (part 2)</td>
<td>128</td>
</tr>
</tbody>
</table>
Dedicated to my grandmother Pina.
Chapter 1

Introduction
Chapter 1: Introduction

The Internet has made possible a wide spectrum of distributed cooperative applications in several areas, such as collaborative e-commerce [28], distance learning, telemedicine, e-government. A requirement common to many cooperative application environments is the need for secure document exchange. By secure exchange we mean that document confidentiality and integrity are ensured when documents flow among different parties within an organization or within different organizations. Ensuring document confidentiality means that document contents can only be disclosed to subjects authorized according to access control policies agreed upon by the various parties. Ensuring document integrity means that the document contents must be correct with respect to a given application domain and that the document contents may be modified only by authorized subjects. It is a common case in many application environments that not all parties are authorized to modify any document that is exchanged among these parties. Rather, different parties can be given selective update privileges to different documents, or even different components of the same document. Whereas the problem of documents confidentiality has been widely investigated [37], the problem of how to ensure that a document, when exchanged among different parties, is modified only according to the stated policies still lacks comprehensive solutions. We believe that such a comprehensive solution requires:

1. A model and a high-level language for specifying update policies - such a model and language are crucial whenever several parties need to state commonly agreed-upon policies according to which documents can be modified by the involved parties.

2. An infrastructure supporting the specification and enforcement of such policies in a distributed environment.

In this thesis, we present some approaches to realize such a comprehensive solution. We assume that documents to be protected are encoded in XML [35]. We have chosen to cast our approach in the framework of XML documents because of the widespread adoption of such a document standard in a large variety of application environments. However, we believe that our approach can be easily extended to other document exchange formats.

The key ingredients of our approaches can be summarized as follows. We provide an access control model supporting, besides several document browsing privileges, various authoring privileges, such as privileges for deleting and modifying document elements and attributes, or inserting new elements and attributes into documents. These authorization privileges support a fine granularity level of control on document modifications. An important aspect of our access control
model is the use of subject credentials. A credential is a set of properties concerning a subject that are relevant for security purposes (for example, the position of the subject within the organization, projects a subject is working on, etc.). Authorizations are then expressed by specifying the subjects receiving the authorizations in terms of conditions against the subject credentials. Subject credentials thus represent a way to support access control based on subject qualifications and profiles. In our model, both credentials and update policies are encoded in XML. Therefore, we not only provide a high-level language for policy representations, but we can also apply the protection mechanisms we provide for regular XML documents to credentials and access control policies. Such a capability is crucial in an environment where credentials and access control policies themselves need to be exchanged among the various parties, for purposes such as access control policy negotiations.

Our access control model is complemented by an infrastructure supporting secure cooperative document updates. The basic idea underlying our approaches is that the document originator (do), that is the subject that generates the document to be updated and determines who can modify it and according to which mode, sends the document to be modified to a given subject; this subject operates on the document and then forwards it to a second subject and so forth. Each subject upon receiving the document from the do or from the previous subject along the path must be able to modify all and only those portions of the document for which it has a proper authorization according to the specified security policies. The main goal of our approaches is to enable a subject, upon receiving a document, to verify whether the updates performed on the document up to that point are correct with respect to the stated policies. Our approaches are based on the use of hash functions and digital signatures.

Another key requirement is not only that of ensuring document integrity and authenticity, or to ensure that documents are modified according to the access control policies agreed upon by the various parties, but also that, if required by the application at hand, the documents be actually transmitted to the parties according to a specified order. By a specified order, we mean that a specification is associated with the transmitted document stating the order according to which the document must be received by the involved parties. Such specification can be very rigid, that is, it can consist of the full list of all the subjects having to receive the document, or can be more flexible by allowing a partial list, containing only some of the subjects having to receive the document. The involved subjects form a so-called Cooperative Group (CG), that is, a set of subjects to which the document can be sent to be read and/or updated. Possible specifications may also differ with respect

---

1By subject we mean either a human user or a software application.
to how subjects are identified; one may want to specify subjects by using subject identifiers as well as by using roles or credentials. In the approaches proposed in this thesis we make use of different specifications of the path that a document must follow during a distributed and cooperative update process.

The work presented in this thesis has been developed in the framework of the Author-\(\lambda\) project [4]. Author-\(\lambda\) is a Java-based system for access control and security policy design for XML documents. For access control, Author-\(\lambda\) supports credential-based policy specifications at varying granularity levels. Additionally, Author-\(\lambda\) supports push and pull distribution policies for document release. A number of administration tools are also provided, to facilitate security administration according to the underlying security policies. What we describe in this thesis are the techniques and protocols provided by Author-\(\lambda\) to enforce distributed document updates. This is a major extension since it requires, besides an extension to the policy specification language, the development of an infrastructure and related algorithms for supporting correct update operations in a distributed environment in which subjects can autonomously verify the correctness of update operations without interacting, in most cases, with the do. These features were not supported by the previous versions of Author-\(\lambda\) and, to the best of our knowledge, they have not been proposed before.

In this thesis we propose three approaches to achieve the goal of making possible a distributed and cooperative update process.

The first approach is particularly suited for non-colluding byzantine distributed systems, that is distributed systems in which there are some subjects, involved in the update process, that do not obey to the stated protocol, but they do not collude. This approach proposes the addition to the XML document of control information used by the protocol to check the document content integrity. With respect to specification of the path that the document must follow this approach makes use of subject identifiers to identify subjects that will take part to the update process.

The second proposed approach is particularly suited for colluding byzantine distributed systems. This means that it proposes some mechanisms to detect if one or more byzantine subjects have illegally modified the document content and/or the path that the document must follow. Also in this approach subjects in the path are identified through subject identifiers. This approach also makes possible to build not only the last version of the document content, but also all the previous versions. This approach offers a higher security level according to the possibility of detecting corruption of the document/path content wrt the first proposed approach, but it requires to attach to the document a larger amount of information.
Chapter 1: Introduction

The third approach we propose is particularly suited for colluding byzantine and failure prone distributed systems. This means that failures of do not affect the process. Moreover this approach requires the presence of some special subjects, called delegates, that substitute the do during the update process for the execution of some functions, such as for example the document content recovery. Also the failure of some delegates, during the process, does not affect the process itself. Another feature provided by this approach is the possibility of detecting malicious operations executed by some colluding byzantine subjects involved in the update process or by some colluding byzantine delegates, according to some parameters set at the beginning of the process. Even though this approach requires a more complex infrastructure wrt the previous approaches, it provides the above-mentioned additional and innovative advantages not provided by the other approaches.

The remainder of this thesis is organized as follows. Chapter 2 introduces work related to the approaches proposed in this thesis and compares it with our proposals. Chapter 3 briefly summarizes basic concepts of XML and the access control model on which our infrastructure for distributed update relies. Chapter 4 presents our first approach for distributed and collaborative updates particularly suited for non-colluding byzantine distributed systems, whereas Chapter 5 presents the second approach dealing with colluding byzantine distributed systems. By contrast, Chapter 6 presents our third approach dealing with colluding byzantine and failure prone distributed systems. Chapter 7 concludes the thesis and outlines future work. Appendix A presents the formal proofs of the results presented in the thesis.
Chapter 2

Related work

Contents

2.1 Introduction ............................................. 7
2.2 Access Control Models and Mechanisms ..................... 7
2.3 XML Document Updates .................................. 10
2.4 Encryption and Digital Signature Techniques ................. 11
2.5 Delegation of Authorizations ............................. 12
2.6 Collaborative Computing ................................. 14
2.7 Group Communication ................................. 15
Chapter 2: Related work

2.1 Introduction

Several research groups from both academia and industry are currently investigating problems related to security and XML. An overview of research work and commercial products related to XML security can be found in [37]. To the best of our knowledge, the work reported in this thesis is the first to propose a comprehensive approach to the problem of distributed and cooperative updates of XML documents. Even though we are not aware of other proposals to which our model can be directly compared, the access control model on which our infrastructure relies has some relationships with access control models and mechanisms developed for object-oriented DBMSs [9, 20], HTML documents [24] and XML documents [6]. Additional related work includes the XACML and SAML standards proposed by OASIS [23, 34], and proposals regarding the update of XML documents [29, 12, 11]. To develop collaborative and distributed update techniques for XML documents, we have borrowed some ideas from existing work dealing with the areas cited in the following, extending them with respect to our goals and proposing mechanisms to assure integrity, authenticity and confidentiality of document portions exchanged during the update process that, to the best of our knowledge, have not been proposed by any other previous work. Additionally, the management of distributed updates of XML documents requires the use of cryptographic methods to ensure confidentiality of the exchanged contents. Furthermore, collaborative updates has some relations with work proposing methods to realize collaborative computing [18, 7, 19], and the distributed nature of the proposed update process implies also a relation with the area of group communication and fault tolerance [31, 22, 21].

For this reason, in what follows we briefly review work in all those areas and compare them with our work.

2.2 Access Control Models and Mechanisms

In this section we discuss some work, related with this thesis, dealing with the specification of access control models and mechanisms. In particular the models proposed in [9, 20] are specifically tailored to an object-oriented DBMS storing conventional, structured data. As such, great attention has been devoted to concepts such as versions and composite objects, which are typical of an object-oriented context and do not apply to the XML environment. Like our access control model, those models support the concept of authorization both at the schema and instance level, even if our model has a larger variety of authorization propagation options in that it supports
three different options by using which the Security Administrator (SA) can specify: \(i\) that an author-
ization defined at a given level in the XML document hierarchy propagates to all lower levels; \(ii\) that the propagation stops at the first level down in the hierarchy; or \(iii\) that no propagation has to be enforced. By contrast, ORION [20] has only one propagation policy, which is equivalent to
the first option. This thesis also proposes a credential based access control model, that is, subjects to which authorizations are granted are specified according to the properties they hold as stated in
their credentials.

Other related approaches has been carried for WWW documents. For instance an access control model for WWW documents has been proposed in [24]. In such a model, HTML documents are organized as unstructured pages connected by links. Authorizations can be granted either on the whole document or on selected document portions. Although we borrow from [24] the idea of selectively granting access to a document (by authorizing a subject to see only some portions and/or links in the document), our work substantially differs from this proposal. Differences are due to
the richer structure of XML documents with respect to HTML documents and to the possibility of attaching a DTD/XMLSchema to an XML document, describing its structure. Such features require the definition and enforcement of more sophisticated access control policies than the ones devised for HTML documents. The access control model proposed in [24] has great limitations deriving from the fact that it is not based on a language able to semantically structure the data, as in our model for XML. As such, authorization administration is very difficult. In particular, if one wants to give access to portions of a document, it has to manually split the page into different slots
on which different authorizations are given. This problem is completely overcome by our model because XML provides semantic information for various document components. Authorizations can thus be based on this semantic information. Moreover the access control model proposed in [24] does not take into account the problem of documents updates, that is widely investigated in this thesis in particular with respect to distributed and collaborative systems.

An access control model for XML documents has been proposed in [6]. In particular, this model does not consider the problem of a secure massive distribution of XML documents since it considers only the information pull mode for document distribution. The model proposed in [6] provides a fully description of the read access mode, but it does not provide an articulated set of au-
thorization privileges as our model does. More precisely, the model in [6] considers write operations only on single nodes and provides a mechanism to evaluate correctness of the executed operations according to the constraints associated with the specified policies. By contrast, we provide a number of specialized access modes for browsing and authoring, which allow the SA to authorize a subject
to read the information in an element and/or to navigate through its links, or to modify/delete the content of an element/attribute. Moreover our model deals with distributed updates of XML documents, whereas related work [6] is tailored for centralized systems in which access to an XML document is granted according to the request sent by a subject, that is evaluated with respect to a set of access control policies stating the rights possessed by requiring subjects.

An extensible access control markup language (XACML) has been recently proposed as OASIS standard [34]. There are two main differences between this language and the one on which our model relies. XACML supports the concept of authorization propagation only for the request specification, but it does not support this feature for policy specification. By contrast, our language supports several authorization propagation options for policy specification as already mentioned. Moreover, XACML supports the concept of subject’s role [26], whereas our model is based on the more general concept of subject’s credential [32].

A security assertion markup language (SAML) has also been recently proposed as OASIS standard [23]. This language supports the specification of authorization requests and responses to be exchanged during a distributed transaction to obtain, for example, authorization or authentication assertions that a subject needs to verify in order to access a particular resource. The architecture underlying SAML relies on three parties: a subject $S$, that wishes to access a particular resource $r$, a Relying Party $RP$, that manages the requested resource and needs to authenticate $S$ to grant/deny access to $r$, and an Asserting Party $AP$ that can assert that $S$ has been authenticated. SAML has a very general notion of protection object in that a protection object is generically a resource, whereas our model is specifically tailored to XML documents. As such SAML is not able to support several features that are relevant to the protection of XML documents. Moreover SAML supports the following types of actions to be exercised on a resource: Read, Write, Execute, Delete and Control; whereas our model supports a greater and more specific set of privileges to be exercised on XML documents, allowing a subject to modify both elements and attributes. Furthermore, our work includes not only the definition of a language but also the development of a system, able among other things to support certified distributed updates. Note, however, that our approach to enforce secure distributed updates to XML documents can be used also when different document models and update authorization languages are adopted. A major difference of the proposed approaches to distributed and cooperative updates of XML documents, presented in this thesis, with respect to SAML is that our approach uses a decentralized mechanism to locally check the satisfaction of receiver profiles, composing a flow policy, and the authenticity of receiver information, without contacting any other party.
2.3 XML Document Updates

Methods for updating XML documents have been proposed in [29, 12, 11]. In particular [29] proposes a set of primitive operations for modifying the structure and content of an XML document, an extension to the XQuery language [40] proposed by W3C [33] in terms of commands expressed in an SQL-like format to deal with document updates, and, finally, algorithms for dealing with update of XML documents/views stored within a relational database. This proposal takes into account only technical aspects regarding how to realize XML document updates, without dealing with the problem of granting document update rights to a subject according to a set of stated access control policies as this thesis provides.

An access control model and a technique that supports not only read operations but also update operations has been proposed in [12]. Privileges supported by the proposed model are: insert, delete, replace, rename, and read. The model associates with each privilege a type information which is applied to a particular element in a XML document, according to its definition in the corresponding DTD. This information is used to distinguish operations that modify the structure of an XML document, instance of that DTD (\(D\) type operations) from those that only modify the XML document content (\(U\) type operations). This distinction is made both according to the structure of the XML document and to the number of elements effectively present in the XML document itself. For example, the insertion of an element associated with symbol (?) in the DTD, already present in the XML document is considered a \(D\) type operation, because it changes the structure of the XML document. By contrast, the same insertion is considered a \(U\) type operation if that element is not present in the XML document, because this insertion respects the XML document structure and changes only its content. The model executes a two steps process to determine if a subject can execute a requested operation on an XML document. In the first step, the request is analyzed and if the required operation is of \(D\) type and the subject can only execute \(U\) type operations, according to the stated access control policies, on the specified XML document, the request is rejected. Otherwise, in the second step the model checks if the elements, belonging to the XML document required to be modified, are really modifiable according to the labeling of the XML document. The labeling is done by adding a label, indicating the type of operation that can be executed, to each element of the document that is accessible by that subject according to the stated access control policies. Since the specification of the access control policies provide the possibility of generating both positive and negative policies, an element of the document can result accessible (positive label) or not accessible (negative label) accordingly. This two steps method is proposed...
to reduce the cost of access determination in those cases in which (step 1) the access requirements are not satisfied at operation type level. Indeed in those cases the proposed method does not require to produce the document labeling. With respect to the proposal of this thesis this work does not take into account the distributed update problem and does not make use of credentials to specify the subjects to which a privilege is granted as is done in our proposal.

In [11] a rewriting method, called SAXE, is proposed to generate safe Update-XQuery statements starting from corresponding Update-XQuery statements. This method given an update query formalized with XQuery language, translates it into another modification query expressed in XQuery language, containing additional conditions (constraints) determined according to the XML Schema associated with the XML document to be updated by the original update query. This method assures that only update queries that generate XML documents valid with respect to the corresponding XML Schema are applied to the original XML documents, thus preserving validity. Also in this case security issues, such as access control of the resources, are not taken into account, neither is the distributed update problem.

In general, with respect to our work, the above-mentioned proposals focus on the method and mechanism used to modify XML documents, whereas we present an infrastructure needed to allow subject to update XML document in a distributed system mainly preserving integrity of the exchanged contents. Thus, proposals in [29, 12, 11] give an answer to the question: "How is it possible to update XML documents?", whereas this thesis tries to give an answer to the question: "How can subjects cooperatively update an XML document in a distributed and secure system?"

### 2.4 Encryption and Digital Signature Techniques

Managing distributed and collaborative updates of XML documents entails also the use of an encryption method to assure confidentiality of the exchanged content and of digital signature techniques for authentication purposes. W3C proposes several solutions to accomplish the encryption task as described in the XML Encryption Syntax and Processing Recommendation [36], and also to generate as well as XML digital signatures, as described in the XML Signature Syntax and Processing Recommendation [38]. In particular, XML encryption allows one to encrypt an entire element and all the elements it contains, only the elements contained in an element, the content (Character Data) contained in an element, or an arbitrary data and whole XML documents. In this thesis we have generated the encryption of the XML document portions according to this recommendation, applying among the W3C proposed modalities that one that fits better our confi-
dentiality purposes. In particular in this thesis we have used the encryption of an entire element (its sub-elements included) for the encryption of the content associated with an atomic element, that is, the minimum XML document portion that can be accessed and modified and thus independently encrypted and protected, and also for the encryption of the position information associated with each atomic element, indicating the position of the atomic element content within the original XML document used in the integrity checking phase and in the view generation phase.

XML Signature Syntax and Processing Recommendation provides several types of digital signature, such as: enveloping signatures, that is, signatures that have as their children the objects to be signed; enveloped signatures, that is, signatures contained in the object to be signed; and finally detached signatures, that is, signatures that allow one to sign objects in the same document or XML document portions within external XML documents. In this thesis we have adopted both the enveloping and enveloped signatures to authenticate reserved information and to certify modification rights possessed by subjects.

### 2.5 Delegation of Authorizations

Another important aspect to be taken into account when dealing with distributed updates of XML documents is the distribution of modification rights to subjects involved in the update process, often called delegation of authorizations. It is required that each subject knows which privileges it can exercise and on which portion of an XML document, and also that a subject can check if another subject has operated on a document portion according to the rights it really possesses with respect to that portion. A lot of work has been carried out in this area, in particular [2, 10, 30].

For instance [2] deals with the specification of entities that receive some authorizations on a particular resource by the owner of the resource itself, and the possibility of granting partially or totally to another entity the received authorizations generating an authorization delegation chain. In particular in [2] public keys are used as identifiers for the entities receiving authorizations, and we borrow this idea to univocally identify subjects involved in the update process, associating also this information with the credentials possessed by subjects in order to allow a subject to check the correct possession of some required requirements by a particular subject, specified in terms of the corresponding public key. An important difference between this work and our proposal is that in [2] entities that receive authorizations uses them to compose access requests to be directly sent to the owner, whereas in our proposal a subject uses a received authorization to guarantee subsequent subjects that it possesses a valid right to modify a particular document portion.
In [10] a practical technique for delegation is described, and a mechanism for delegation termination, thus to ensure that a delegated party $dp$ cannot indefinitely act on behalf of the subject that has generated the delegation for $dp$ itself. This proposal makes use of certificates generated by the owner of a privilege to grant the same privilege on a resource to third delegated parties. These certificates are signed by the owner and contain the public key of a delegated party. Only this party will be able to use a certificate, to gain the access described in the certificate itself, by proving that it possesses the certificate and the contained public key belongs to this delegated party. In this thesis we use the same mechanism to guarantee a subject that another subject has the proper right to execute an operation on some document portions, and to allow a subject to know which privileges it can exercise. In [10] there is also the possibility of generating a delegation chain. In this thesis we do not make use of this mechanism, thus subjects can only receive privileges they can exercise on document portion against stated access control policies. Only in the third of our proposed update approaches there is a mechanism that allows a subject to delegate another subject the possibility of granting/denying a privilege (flow policy extension) on behalf of the subject who originated that flow policy.

In [30] a mechanism is proposed supporting distributed accesses to remote resources. These resources are managed by some resource managers that must accomplish the task of evaluating access requests from users for the managed resources. Policies to regulate access to the managed resources are stated by the owners of the resources themselves. In particular resources can be owned by the resource managers themselves or by third parties. The proposed system is based on the use of certificates. In particular two types of certificates are used: use-condition certificates and attribute certificates. The first certificate type states conditions according to which a user can gain access to a required resource, whereas the second certificate type states information concerning the identity of a user. The proposed approach defines a mechanism that initially checks the identity of the requesting user according to the presented attribute certificates, then the resource manager checks if there is at least a use-condition certificate for the authenticated user for the requested resource to grant/deny access to the resource. In this thesis, we also make use of certificates to manage accesses to resources. By contrast, in our context there is not a unique entity that check if a modification access to a particular resource is correct, but all subjects participating to the update process acts as a resource manager. Moreover, our model provides a cryptographic access control mechanism, that is, the document to be updated is entirely encrypted and only subjects authorized to access a particular document portion obtain the corresponding decryption key needed to gain access to the associated clear text.
2.6 Collaborative Computing

Collaborative computing is a process in which a group of parties jointly work to reach a common goal. Some proposals in this area are [19, 18, 7]. More precisely, [19] proposes an approach to the representation, maintenance, and enforcement of policies, establishing who can use which resources and for which purposes among the parties belonging to a collaborative community. This proposal also realizes a separation between the owner of the resources to be accessed and the community that requires to use them, allowing the owner to grant a set of authorizations to a collection of resources directly to the community, and delegating the fine-grained access control to the community itself. Indeed, a user belonging to the community has to send an access request for a resource to a Community Authorization Service (CAS) server, entitled to evaluate these requests, that according to the policies stated by CAS itself returns some capabilities to the user. The user then presents these capabilities to the owner of the resources, that according to this information grants or denies the required access. [18] proposes a Collaborative Computing Data Space (CCDS), where shared resources are accessible by members of a collaborative community according to the permissions stated by the owner when the particular resource has been created or imported in CCDS. Resources can be files, directories, pipes, etc, and permissions over them are those typically associated with objects in a Unix environment, that is, read, write, and execute. Resources are shared among all the participants of the community that can access them according to the stated permissions.

[7] proposes a method to realize a collaborative environment, called collaboration, that consists of a group of principals sharing messages and/or files and that can add new members to the group. In particular in [7] a mechanism is proposed to better bind a user with the corresponding public key, giving also the possibility of associating a familiar name with the identified person. The process of identification thus takes into account not only digital aspects, such as public key verification, but also biometric checks made through image/voice recognition of the counterpart. A user and then its corresponding public key are thus associated with a mental recognition process and this method allows a subject to correctly establish this binding in presence of a large number of users with the same name claiming to be the user with which a principal wants share some information. This proposal allows also a principal, called the creator of a collaboration, to add new members to a group, possibly giving to them the possibility of adding new members themselves. Each member in the collaboration can access all the resources made public by a member. With respect to the collaborative update of XML documents proposed in this thesis, we define an access control model
that allows the subject, corresponding to the collaboration creator, to exactly specify who can access which objects and according to which access mode, generating a finer grained access control over the objects (document portions) to be protected. We borrow the idea of a collaboration creator that starts the collaboration, and the possibility of adding new members to the collaborative group, and also the possibility for those members, if specified by the creator, to add by themselves new members by using an ad-hoc language for the flow specification. This language beyond allowing these specifications, gives also a subject the possibility of stating a desired order in which document portions must be read and/or modified by collaborative group members.

2.7 Group Communication

The distributed nature of the collaborative update of XML documents with respect to the last mode proposed in this thesis implies, as a requirement, the use of group communication techniques. In the literature group communication is classically associated with the concept of view of a communicating group of subjects. In particular, a view represents the set of subjects currently recognized to be members of the group and so entitled to exchange messages with each other member of the group. Mechanisms to add new members and/or to remove members from the current view are also provided. A survey of the main group communication specifications is given in [31]. In our last proposed approach to XML document updates we adopt a group management technique which does not require updating the current view of the group, considering instead in each instant the initial whole communicating group chosen by the subject that has started the process. The proposed protocols require a subject to always send a message to all the members of the group, managing accordingly the situations in which one or more members are down. Thus, these protocols take into account the fault tolerance problem inherent in the asynchronous and failure-prone nature of distributed systems. Their construction has been heavily influenced by protocols proposed in [22, 21].

Another feature of these protocols is that of providing methods to overcome malicious behaviors of a limited number of members, called byzantines. The byzantine problem has been extensively investigated [13, 17, 15, 14, 16]. Most approaches are based on the specification of conditions according to which it is possible to detect malicious behaviors of byzantines and to continue without affecting the global computation. In this thesis we have taken into account this problem in all the three proposed approaches. We borrow from the above mentioned proposals the idea of adding a number of redundant subjects in a communicating group to prevent the supposed number
of byzantine subjects in the group from affecting the communication protocol with their behaviour. Moreover we borrow the idea that when dealing with set of entities containing some byzantine ones, each entity must receive a number of messages determined according to the estimated number of byzantines to allow the protocols to correctly make progress.
Chapter 3

A Framework for distributed and cooperative updates of XML Documents

Contents

3.1 Introduction ................................................. 18
3.2 Basic Concepts of XML ...................................... 20
3.3 An Access Control Model for XML Documents ................. 21
3.4 Document Encryption ........................................ 25
3.5 System Implementation ........................................ 29

Figures

3.1 Overview of the update approach ................................ 19
3.2 (a) An example of XML document and (b) its graph representation .... 20
3.3 Examples of $\mathcal{X}$-sec credentials .......................... 22
3.4 An example of Policy Base .................................... 26
3.5 The $\mathcal{X}$-Sec Policy Base template .......................... 27
3.6 Author-$\mathcal{X}$ architecture .................................... 29

Tables

3.1 Access privileges and their semantics .......................... 23
3.2 Table Key info for the document in Figure 3.2 ................. 28

17
3.1 Introduction

Updates to XML documents can be made according to two different modes. Under the first, which is more traditional, a subject wishing to modify an XML document sends a request to the XML document originator (do), that is, the subject that generates the XML document to be updated and determines who can access/modify this document, or its portions, and in which mode. The do, on the basis of the specified access control policies, decides whether the operation can be authorized (partially or totally) or should be denied. However, there can be cases in which this traditional approach is not adequate or it can be inefficient (since it requires an interaction with the do for each document modification). For these reasons, in this thesis we propose some alternative approaches to document updates which rely on encryption and digital signature techniques and support a distributed approach to document updates. We will use in the subsequent chapters the following notations: $E_k(x)$, denoting the encryption of $x$ with the symmetric key $k$, using a symmetric cipher system; and $S_s(x)$, denoting the content of $x$ digitally signed by the subject $s$ with its private key. The idea is motivated by the fact that often, within an organization, XML documents are subject to pre-defined cooperative update processes (which usually take place at specific periods of times) according to which different organizational roles must be able to modify possibly different portions of the same document. Each subject receiving the document must be able to modify all and only those portions of the document for which it has a proper authorization and then it has to pass the document to another subject for additional operations. The idea is to develop a framework supporting this update mode, able to minimize the interactions with the do and, at the same time, guaranteeing the correct enforcement of access and authoring privileges on the document. Such a framework makes use of different keys for encrypting different portions of the same document according to the specified access control policies. Each portion is encrypted with one and only one key. The same (encrypted) copy of the document is then sent to a subject belonging to a Collaborative Group ($CG$), where by collaborative group we mean a set of subjects that may receive the document for updating or reading it. The document before returning to the do must be seen and/or modified by subjects in the $CG$, according to a specified set of conditions, generically called path conditions - for example a manager must be the last subject that receives the document. Each subject in the $CG$ only receives the key(s) for the portion(s) it is enabled to see and/or modify (see Figure 3.1 for a general overview of the approach) and the path conditions. Such conditions together with other criteria are used by the subject in order to determine the next receiver of the document from the set of subjects in the $CG$. The approaches we propose in the subsequent chapters are distributed
in the sense that each subject, under specific assumptions, once receiving the encrypted document, is able to verify, without interacting with the do, whether the operations performed till that point on the document are correct (that is, they do not violate the access control policies specified by the do for that document). This goal is obtained by attaching to the encrypted document some additional control information, with the purpose of making a subject able to verify the correctness of the updates performed so far on the document, without the need of interacting with the do. The encrypted document and the control information form the so called document package. Each subject s also receives some certificates, signed by do, stating which privileges s can exercise on a particular document d. These certificates are used to guarantee subsequent receivers that s possesses the proper right for executing a modification on the d’s content.

The remainder of this chapter is organized as follows. Section 3.2 gives a brief overview of the XML language, whereas Section 3.3 introduces the access control model on which our framework relies. By contrast Section 3.4 describes how we realize document encryption, and finally Section 3.5 concludes the chapter presenting a general overview of our system implementation.
Chapter 3: A Framework for distributed and cooperative updates of XML Documents

3.2 Basic Concepts of XML

Building blocks of XML documents [35] are nested, tagged elements. Each tagged element has zero or more subelements, zero or more attributes, and may contain textual information (data content). Elements can nested at any depth in the document structure. Attributes are of the form \texttt{name = attvalue}, where \texttt{name} is a label and \texttt{attvalue} is a string delimited by quotes. Attributes can have different types allowing one to specify the element identifier (attributes of type ID often called \texttt{id}), additional information about the element (e.g., attributes of typeCDATA containing textual information), or links to other elements of the document (attributes of type IDREF/URI referring to a single target or IDREF(s)/URI(s) referring to multiple targets). An example of XML document is given in Figure 3.2(a). This document is a monthly report produced by a department, containing an overall description portion that gives some general information, a balance sheet variations section with items under hardware and software, and an approval overall descr section.

Figure 3.2: (a) An example of XML document and (b) its graph representation.
sheet variation that specifies new values concerning some items, and finally an approval portion containing some notes.

Based on this nested structure, an XML document can be represented as a graph, as illustrated in Figure 3.2(b). In the graph representation, white nodes represent elements, whereas gray nodes represent attributes. A node representing an element contains the element identifier (id). An element identifier can be the ID attribute value associated with the element, or can be automatically generated by the system, if no attribute of type ID is defined (system defined id are represented as &n, where n is a natural number). A node representing an attribute contains its associated value. For simplicity, the data content of an element is represented as a particular attribute whose name is content and whose value is the element data content itself. The graph can contain edges representing the element-attribute and the element-subelement relationships, and link edges, representing links between elements introduced by IDREF/URI attributes. Edges are labeled with the tag of the destination node (i.e., an element or an attribute) and are represented by solid lines, whereas link edges are labeled with the name of the corresponding IDREF/URI attribute and are represented by dashed lines. A document type declaration can be attached to XML documents, specifying the syntactic rules that XML documents must follow. These rules are collectively known as the Document Type Definition (DTD) or XML Schema. An XML source is a set of XML documents and associated DTDs/XML Schemas. Throughout the thesis, we assume that an XML source S is given.

3.3 An Access Control Model for XML Documents

In this section we briefly review the access control model on which the proposed infrastructure relies. We first characterize how subjects are qualified in access control policies. Then, we introduce the concept of protection object, and the access privileges supported by the model. Finally, we introduce propagation options and we show how all the above-mentioned components are used in the specification of access control policies.

Subject. To better take into account subject profiles in the formulation of access control policies, subjects are qualified by means of credentials. A credential is a set of characteristics, that describe a subject, important for security purposes (e.g. age, position within an organization). For instance, by using credentials, one can simply formulate policies such as “Only programmers that are permanent staff can access documents related to the internals of the system”. To make easier the process of credential specification, credentials with similar structure are grouped into credential
<manager s_id="154" cid="202">
  <RSAKeyValue>PK_{154}</RSAKeyValue>
  <name>
    <Fname>Bob</Fname>
    <Iname>Watson</Iname>
  </name>
  <age>39</age>
  <department>R&D</department>
  <salary>8,000</salary>
  <category>Top Executive</category>
</manager>

<secretary s_id="102" cid="232" manager="154">
  <RSAKeyValue>PK_{102}</RSAKeyValue>
  <name>
    <Fname>Tom</Fname>
    <Iname>Moore</Iname>
  </name>
  <age>25</age>
  <department>R&D</department>
  <salary>2,000</salary>
  <level>third</level>
  <duty>manager secretary</duty>
</secretary>

<accountant s_id="104" cid="132">
  <RSAKeyValue>PK_{104}</RSAKeyValue>
  <name>
    <Fname>John</Fname>
    <Iname>Brown</Iname>
  </name>
  <age>45</age>
  <department>R&D</department>
  <salary>6,000</salary>
  <level>third</level>
</accountant>

<company_management_director s_id="146" cid="224">
  <RSAKeyValue>PK_{146}</RSAKeyValue>
  <name>
    <Fname>Alice</Fname>
    <Iname>Smith</Iname>
  </name>
  <age>50</age>
  <department>R&D</department>
  <salary>20,000</salary>
  <category>Top Executive</category>
</company_management_director>

<notary s_id="138" cid="304">
  <RSAKeyValue>PK_{138}</RSAKeyValue>
  <name>
    <Fname>Mary</Fname>
    <Iname>Flynn</Iname>
  </name>
  <age>54</age>
  <law_firm>FLYNN & FLYNN</law_firm>
</notary>

Figure 3.3: Examples of X-sec credentials
types. We encode both credentials and credential types using an XML-based language, called \( \mathcal{X} \)-Sec [3]. Figure 3.3 shows an example of \( \mathcal{X} \)-Sec credentials. Each subject has one or more credentials issued by several Certification Authorities (CAs). Whenever a CA generates a credential for a subject \( s \) that CA inserts in the credential also the public key of \( s \), to prevent other subjects different from \( s \) from using that credential. Access control policies specify conditions against credentials and credential properties. These conditions (which are expressed by means of an XPath-based language [39]) implicitly identify the set of subjects to which a policy applies. Examples of conditions are: *All top executive managers*, or *All secretaries working at the R&D Department.*

**Protection objects.** By protection object we mean the entities to which an access control policy applies. The model provides a wide range of protection objects, in that it is possible to specify policies that apply to: *i*) all the instances of a DTD/XML Schema; *ii*) collections of documents; and *iii*) selected portions within a document(s) (i.e., an element (or a set of elements), an attribute (or a set of attributes), a link (or a set of links)). This wide range of protection objects is complemented by content-dependent access control, that is, the possibility of specifying access control policies based on document content in addition to document structure. The protection objects to which a policy applies are specified through an X-Path compliant language.

<table>
<thead>
<tr>
<th>Type</th>
<th>Privilege</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Browsing</td>
<td>view</td>
<td>To read the values of all the attributes in a protection object, apart from attributes of type IDREF(s)/URI(s). The view privilege can also be given on selected attributes within an element</td>
</tr>
<tr>
<td></td>
<td>navigate</td>
<td>To see all the links implied by attributes of type IDREF(s)/URI(s) contained in a protection object. The navigate privilege can also be given on selected attributes within an element. The view the subject has on the referred elements depends on the authorizations the subject has on them.</td>
</tr>
<tr>
<td>Authoring</td>
<td>delete_attr</td>
<td>To remove an attribute from an element</td>
</tr>
<tr>
<td></td>
<td>insert_attr</td>
<td>To add an attribute to an element</td>
</tr>
<tr>
<td></td>
<td>update_attr</td>
<td>To modify an attribute value</td>
</tr>
<tr>
<td></td>
<td>insert_elmt</td>
<td>To insert new elements that are direct subelements of the element on which the insert_elmt privilege is specified</td>
</tr>
<tr>
<td></td>
<td>delete_elmt</td>
<td>To remove the subtree rooted at the element on which the delete_elmt privilege is specified</td>
</tr>
</tbody>
</table>

Table 3.1: Access privileges and their semantics

**Privileges.** Access control policies can be categorized into two groups: *authoring policies*, that allow a subject to modify a protection object, and *browsing policies*, that allow a subject
to access the information contained into a protection object. Two browsing privileges are supported: view and navigate, that allow subjects to read the information in a protection object and/or to see the relations occurring among protection objects (defined through IDREF(s)/URI(s) attributes). Authoring privileges allow subjects to modify/delete or insert protection objects. We support five authoring privileges: three at the attribute level (delete\_attr, insert\_attr, and update\_attr) and two at the document/element level (insert\_elemt and delete\_elemt). The semantics of access privileges is given in Table 3.1.

**Propagation options.** A further distinguishing feature of our access control model is that a set of propagation options can be exploited in the specification of access control policies. Propagation options specify whether and how a policy specified on a given protection object \( o \) propagates to protection objects that are related to \( o \) by some sort of relationship. Propagation options are therefore a means to concisely express a set of security requirements. Two different types of propagation are provided: implicit and explicit propagation. Implicit propagation is always applied by default and is based on the following principles: 1) policies specified on a DTD/XML Schema automatically propagate to all DTD/XML Schema instances; 2) policies specified on a given element automatically propagate to all the attributes of the element. In addition to implicit propagation one can state whether and how a policy specified on a given protection object propagates to lower level protection objects (wrt the document/DTD/XML Schema hierarchy). Three different options are provided for explicit propagation by means of which can be specified that: i) no propagation is enacted (NO\_PROP option), that is, the policy only applies to the protection objects which appear in its specification; ii) the policy propagates to all direct sub-elements of the elements in the specification (FIRST\_LEVEL option); iii) the policy propagates to all the direct and indirect sub-elements of the elements in the policy specification (CASCADE option). Like credentials, access control policies are encoded using \( \mathcal{X}\-sec \). We denote with the term Policy Base \( (P\_B) \) the XML file encoding access control policies of the source \( S \).\footnote{We assume that each policy is uniquely identified by an identifier, generated by the system when the policy is specified.} Moreover the Policy Base is specified according to the \( \mathcal{X}\-Sec \) Policy Base template presented in Figure 3.5.

Example 1  Figure 3.4 shows a policy base referring to the XML document in Figure 3.2 and generated according to the template reported in Figure 3.5. According to the policies in Figure 3.4 secretaries, managers and accountants working in the R&D department are entitled to see, respectively, the information contained in the monthly report of their department, apart from balance sheet
variations, all information in the monthly report of their department, and only the balance sheet variations. Moreover, secretaries can also modify the overall description part, managers are entitled to update the approval part, and accountants can modify, insert new sub-elements and delete the balance sheet variations element, and delete one of its items. Finally, the company management director can see the monthly reports of all the company departments.

3.4 Document Encryption

A trivial solution for generating the encryption of a document $d$, denoted in the following as $d^e$, able to support our approach is to encrypt the document at the finest granularity level, that is, to encrypt each attribute and element of the document with a different key. This solution, although very easy to implement, may require the generation and distribution of a very large number of keys. To limit the number of keys that need to be generated, we have adopted an alternative approach in which the portions of a document to which the same policies apply are encrypted with the same key. This ensures that it is always possible to deliver to each subject all and only the keys corresponding to the portions of the documents for which it has an authorization, minimizing at the same time the number of encryption keys to be generated. Here we do not go into the details of the techniques developed to support this strategy and we only give the intuition behind them. We refer the interested reader to [5] for further details.

The encryption of a document consists of two main phases: the first, called marking phase, marks each protection object in the source with the identifiers of the applicable policies, whereas in the second phase the document is encrypted based on the results of the first phase. Marking can apply not only to whole protection objects (i.e., attributes and/or elements), but also to the start and end tags of an element only. This possibility allows one to correctly encrypt elements containing attributes to which different policies apply. As an example consider an element $e$ containing two attributes $a_1$ and $a_2$, and suppose that policies $acp_1$ and $acp_2$ apply to $a_1$, whereas $acp_3$ applies to $a_2$. Thus, the view to be returned to a subject to which both policy $acp_1$ and $acp_2$ apply is equal to element $e$ from which all the attributes different from $a_1$ have been removed, whereas the view to be returned to a subject to which only $acp_3$ applies is the element obtained from $e$ by removing all attributes different from $a_2$. Thus, in the document encryption attributes $a_1$ and $a_2$ must be encrypted with different keys, since different policies apply to them. Additionally, another key must be used to encrypt the start and end tags of $e$ that are to be returned to all the subjects entitled to access an
Figure 3.4: An example of Policy Base
attribute of element \( e \).

This leads to the definition of document atomic element, which is the basic portion of an XML document to which encryption can be applied and on which an authoring privilege can be independently exercised.

**Definition 1 (Document Atomic Element).** Let \( d \) be an XML document in \( S \). The set \( DocAE(d) \) of document atomic elements of \( d \) is defined as follows: 1) for each element \( e \) in \( d \), and for each attribute\(^2\) \( a \) in \( e \): \( e.a \in DocAE(d);^3 \) 2) for each element \( e \) in \( d \), \( e.tags \in DocAE(d) \).

While an attribute corresponds to a single portion of a document \( d \) (the attribute name and its value, or only the value for data content), elements consist of two or three non-contiguous components depending on the type of the element. Empty-elements, that is, elements of the form \((<\text{tag-name} \ldots />)\) consist of two components: the first part of the tag name (“\(<\text{tag-name}\)” and its end (“/>”). All other elements consist of three components: the first part of the start-tag (“\(<\text{tag-name}\)”), its end (“>”), and the end-tag (“\(<\text{tag-name}>\)”). This information is important because for each document atomic element \( docae \) it is necessary to define where \( docae \)’s components are located in the original document.

**Example 2** Example of document atomic elements in the XML document in Figure 3.2 are:

\( a) \) \&1.Date corresponding to: “date = “10/1/2003””;

\( ^2\)We also consider in the following the data content associated with an element as an attribute. We denote with the term “dc” this particular type of attribute.

\( ^3\)Here and in what follows we use the dot notation to denote a component of a given structure.
b) &8.content corresponding to: “10K”;
c) &5.tags corresponding to: “<item” “>” “<item>”

A marking for a document $d$ is thus a set of pairs $(docae, P)$, where $docae \in DocAE(d)$, and $P$ is a set of access control policy identifiers. The encryption algorithm groups atomic elements with the same marking and generates a different encryption key for each distinct group, which is used to encrypt all the members of the group. To limit as much as possible the size of the information that circulates among subjects, the encrypted document $d$, delivered to the various subjects, consists only of the encryption of the marked document atomic elements and does not contain non marked components of the document, since these components are not accessible by any subject. The set of document atomic elements which are encrypted with the same key is called document region. We assume that each document region is uniquely identified by an identifier.

In the following, given an XML document $d$ we denote with $DR(d)$ the set of identifiers of the document regions of $d$ implied by the policies in $PB$. Key information are stored into table $Key\_info$ which records, for each document region in a document, the set of document atomic elements that compose the document region, the identifiers of policies that apply to that document region, and the corresponding encryption key.

<table>
<thead>
<tr>
<th>Region</th>
<th>Key</th>
<th>Policies</th>
<th>document atomic elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>K1</td>
<td>{P1, P2, P10}</td>
<td>{&amp;1.tags, &amp;1.Date, &amp;1.Department}</td>
</tr>
<tr>
<td>R2</td>
<td>K2</td>
<td>{P1, P4, P10, P11}</td>
<td>{&amp;4.tags, &amp;4.content, &amp;4.overall_description}</td>
</tr>
<tr>
<td>R3</td>
<td>K3</td>
<td>{P1, P3, P6, P10}</td>
<td>{1.tags, 1.content, 1.ID}</td>
</tr>
<tr>
<td>R4</td>
<td>K4</td>
<td>{P1, P5, P7, P8, P9, P10}</td>
<td>{&amp;3.tags}</td>
</tr>
<tr>
<td>R5</td>
<td>K5</td>
<td>{P1, P5, P7, P9, P10}</td>
<td>{&amp;5.tags, &amp;7.tags, &amp;7.content, &amp;8.tags, &amp;8.content, &amp;6.tags, &amp;9.tags, &amp;9.content, &amp;10.tags, &amp;10.content}</td>
</tr>
</tbody>
</table>

Table 3.2: Table $Key\_info$ for the document in Figure 3.2

**Example 3** Table 3.2 shows the content of table $Key\_info$ associated with the document in Figure 3.2, according to the policies in Figure 3.4.

Our system supports several methods for key delivery [4] and the do can select the most appropriate one depending on the characteristics of the document and of the receiving subjects. Key delivery strategies supported by our system can be classified into two main categories: online and offline. In the online mode both the keys and the package are sent to the subjects by do (together
or separately), whereas in the offline mode keys are stored in an LDAP directory [25] at do and subjects retrieve the necessary keys by querying the directory.

3.5 System Implementation

The protocols described in the subsequent chapters are currently being implemented in the framework of the Author-\(\mathcal{X}\) system [4]. Author-\(\mathcal{X}\) is a java-based system supporting selective, secure and distributed dissemination of XML documents. Its architecture, presented in Figure 3.6, is based on the client-server paradigm. The server system, built on top of the eXcelon XML data server [8], manages all information required for controlling access to documents. In particular the server database is organized in terms of five repositories: the Policy Base (\(P\)\(B\)), the Credential Base (\(C\)\(B\)), the Encrypted Document and Management Information Base (\(E\)\(D\)\(M\)\(I\)\(B\)), the XML Source (\(S\)), and the Authoring Certificate Base (\(A\)\(C\)\(B\)). In addition to the database, the server includes the following main components: \(\mathcal{X}\)-Admin, \(\mathcal{X}\)-Access, and \(\mathcal{X}\)-Update. We briefly describe the first two, and then focus on the last one which implements the protocols defined in this thesis.

The \(\mathcal{X}\)-Admin component provides functions supporting administrative operations, such as for instance specifying or modifying policies, or updating credentials.

The \(\mathcal{X}\)-Access component consists of two subcomponents: \(\mathcal{X}\)-Pull and \(\mathcal{X}\)-Push. The former component supports the selective distribution of the XML documents, stored in the XML Source repository according to the policies in \(P\)\(B\) using the traditional user-on-demand paradigm.
By contrast, the latter component is in charge of supporting document broadcast to user groups. As such, it supports a push-based distribution of the XML documents stored at the server site. In order to support group distribution of the same document and yet to enforce selective access to different components of the document, document components are encrypted by using different keys that are generated according to the stated access control policies. Each subject in the group then receives only the keys for decrypting the document components it can access.

The last component of the Author-X server is X-Update, that manages the collaborative and distributed update process described in this thesis. It generates and updates all control information associated with an XML document. It also generates all certificates associated with a document according to the policies in \( \mathcal{PB} \), and performs the initial steps for the creation and delivery of packages. A package contains the encrypted version of an XML document and the associated control information. Finally, the X-Update module manages the recovery process.

The Author-X client side, called XML-reader, supports several functions for issuing queries and receiving query replies, and for the enforcement of the subject protocols presented in this papers. As shown in Figure 3.6, the XML-reader manages three repositories: DocumentStore (DS), KeyStore (KS), and CertificateStore (CS). The first one records the XML document views obtained as answer to a pull request, or during a server push dissemination of XML documents, or in case of a collaborative and distributed update process. The second repository contains all keys used to decrypt the portions of the XML documents for which the subject has the proper rights. Each key has associated a document and a region identifier to identify which portions of a document can be decrypted using that key. The third repository stores all authoring certificates received from the server.

\[^4\]Encryption is compliant with W3C recommendation [36].
Chapter 4

Cooperative updates in non-colluding byzantine distributed systems

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>33</td>
</tr>
<tr>
<td>4.2</td>
<td>Assumptions</td>
<td>34</td>
</tr>
<tr>
<td>4.3</td>
<td>Generation of Control Information</td>
<td>35</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Authoring Certificates</td>
<td>36</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Control data structures for document regions</td>
<td>37</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Generalized control information</td>
<td>42</td>
</tr>
<tr>
<td>4.4</td>
<td>Subject and Document Originator Protocols</td>
<td>44</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Subject Protocol</td>
<td>44</td>
</tr>
<tr>
<td>4.4.1.1</td>
<td>Package integrity and authentication</td>
<td>45</td>
</tr>
<tr>
<td>4.4.1.2</td>
<td>Package update and delivery</td>
<td>48</td>
</tr>
<tr>
<td>4.4.1.3</td>
<td>An Illustrative Example</td>
<td>49</td>
</tr>
<tr>
<td>4.4.1.4</td>
<td>Correctness of the Subject Protocol</td>
<td>53</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Document Originator Protocol</td>
<td>56</td>
</tr>
<tr>
<td>4.5</td>
<td>Complexity Analysis of the Proposed Approach</td>
<td>57</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Communication cost</td>
<td>58</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Size of exchanged information</td>
<td>60</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Number of certificates generated for a document $d$</td>
<td>62</td>
</tr>
</tbody>
</table>
4.5.4 Execution time of the integrity check protocol and distributed view generation/decryption vs execution time of the centralized view generation/encryption and distributed decryption 

Figures

4.1 Distributed document updates: overall schema ........................................ 34
4.2 Modification of the structure $MR_d$ along an hypothetical document-path .... 43
4.3 Graphic representation of a package ......................................................... 44
4.4 An algorithm for verifying package integrity and authenticity ..................... 45
4.5 Main steps of the package integrity and authenticity algorithm .................. 46
4.6 Update and delivery processes ................................................................. 48
4.7 Update flow of a modifiable region (Step 1) ............................................. 50
4.8 Update flow of a modifiable region (Step 2) ............................................. 51
4.9 Update flow of a modifiable region (Step 3) ............................................. 52

Tables

4.1 Control data structures for non-modifiable regions .................................. 37
4.2 Components of the control data structures for non-modifiable regions ......... 38
4.3 Control data structures for modifiable regions .......................................... 39
4.4 Components of the control data structures for modifiable regions .............. 40
4.5 Parameters involved in the complexity/cost analysis .................................. 58
4.6 Size of the building blocks of a package/access request ........................... 61
4.7 Size of the basic package structures ......................................................... 62
4.8 Time required to perform the basic integrity check protocol operations .......... 65
4.9 Time required to perform the integrity check protocol operations ................ 65
4.1 Introduction

In this chapter we present an approach for the distributed and cooperative updates of XML documents, supported by our framework, particularly suited for non-colluding byzantine distributed systems. The proposed approach is based on a specific infrastructure, that specializes the one presented in Chapter 3. In particular to support the update schema presented in Chapter 3, Figure 3.1, we propose the architecture shown in Figure 4.1 which consists of five main components. The document is first processed by a Parser which, on the basis of the specified policies, analyzes the document structure and groups document portions according to the policies that apply to them. The result is the package structure, containing the document content already grouped according to the above-mentioned strategy together with the control information structure, that is incrementally updated during the package generation process. Such structure contains some control information, that are needed by subjects to verify update correctness. Additionally, it generates a table, named Key Info (see Table 3.2 in Chapter 3), which contains information on the generated groups and their corresponding portions. Both the package structure and the Key Info table are received as input by the Encryption Module, that generates a symmetric key for each group, and stores it in the Key Info table. Then, it encrypts all the document portions with the corresponding keys. The result is the package containing both the encrypted document and the control information structure, and the updated Key Info table. The package is received as input by the Control Information Generator that generates a set of additional control information which are stored in the control information structure. The Dispatcher is in charge of generating the completed document package, containing both the encrypted version of the document and the control information, and of sending it to the first chosen subject belonging to the cooperative group. By contrast, symmetric encryption keys are separately sent to each subject. Finally, the Recovery Manager receives recovery requests from subjects and sends back to the subjects, at the end of a recovery procedure, the last correct version of the package.

The remainder of this chapter is organized as follows. Section 4.2 introduces the assumptions on which the approach proposed in this chapter relies. Data structures required to support distributed updates and document dispatching are covered by Section 4.3, whereas Section 4.4 describes the protocols used by the subjects and the do to check document integrity, and gives an illustrative example of our approach. Section 4.5 reports a complexity analysis of the most relevant operations executed by the proposed protocols.
4.2 Assumptions

It is important to note that our approach relies on a set of assumptions that we discuss in what follows. We assume that each time a subject detects that a portion of the package is inconsistent (that is, a previous subject has operated on that portion violating the policies in $\mathcal{P}_B$) the subject sends a recovery request to the $do$ to obtain the last correct version of the package. This implies that we assume no collusion among the subjects and that, whenever a subject $sbj$ updates a document portion (with proper authorizations and after the execution of the document content integrity check protocol), $sbj$ is sure of the integrity of that portion. Moreover, we assume that a subject sends the package to only one subject in the $CG$, that is, we do not allow a subject to simultaneously send a package to more than one subject. Additionally, to prevent a subject from inserting old versions of document portions into a package we assume that if the receiver $sr$ of the package has already received the package the sender, instead of sending the package to subject $sr$, sends it to $do$ that updates some control information and then sends the package to subject $sr$. This is done to prevent the closure of a cycle in the path followed by the document $d$, which would allow subject $sr$ to insert some portions of old versions of $d$ in the package, without being detected by any other subject. Finally, we assume that each subject knows the public key of all the subjects belonging to the $CG$. 

Figure 4.1: Distributed document updates: overall schema
4.3 Generation of Control Information

After the XML document has been encrypted, as described in Section 3.4 in Chapter 3, the next step is to generate the control information, to be used during the document flow for verifying the correctness of the updates performed on the document. The Control Information Generator module (see Figure 4.1) generates this information for each region of the document and corresponding atomic elements. The generated information differs depending on the access control privileges that can be exercised on a region. For this reason we distinguish between modifiable and non-modifiable regions. Modifiable regions are those whose contents can be modified according to the policies in $P_B$. A region $r$ is thus modifiable if, among the policies that apply to $r$, there exists at least a policy whose access control mode is either delete_attr, delete_elmt or update_attr. By contrast, a non-modifiable region is a region whose original contents cannot be changed according to the policies in $P_B$. Thus, a region is non-modifiable if the access control modes of all the policies that apply to that region belong to the set: \{view, navigate, insert_attr, insert_elmt\}. Note that operations corresponding to insert_attr and insert_elmt privileges alter the document content by inserting a new element and/or attribute; however, unlike the operations corresponding to the delete_attr, delete_elmt, and update_attr privileges, they do not modify the original region, but they can add one or more new regions to the document. Because of this characteristic, they have been inserted among the privileges related to non-modifiable regions. The sets of the identifiers of non-modifiable and modifiable regions of a document $d$ are denoted by $NMR(d)$ and $MR(d)$, respectively.

To enable a subject to verify the integrity of a document content we need different control data structures for non-modifiable and modifiable regions. In particular, the content of the structures for non-modifiable regions is statically defined by the $do$ when the document is delivered to the first subject in the $CG$ and it is not altered by subjects during document transmission. By contrast, the content of structures for modifiable regions dynamically changes according to the updates made on the atomic elements belonging to those regions. In what follows, we refer to policies whose access control modes are in the set $Authoring$-privileges = \{update_attr, delete_attr, delete_elmt, insert_attr, insert_elmt\} as authoring access control policies.

In the remainder of this section we describe in details the control information associated with document regions. Before, presenting this information, we need to introduce the notion of authoring certificate, which plays an important role when dealing with modifiable regions.
4.3.1 Authoring Certificates

Authoring certificates are used by a subject, that has modified a document portion, to prove its right to modify that document portion to the subsequent receivers of the document. Therefore, whenever a subject modifies a document (or a document portion), it has to add the proper authoring certificates to the document control structures. Certificates are generated by the server according to the access control policies in $\mathcal{PB}$. An authoring certificate consists of: an authoring privilege $p$, the id of a subject that can exercise $p$, and the set of atomic elements on which the subject can exercise $p$. Authoring certificates are formally defined as follows.

**Definition 2 (Authoring Certificate).** Let $d$ be an XML document in $S$, and let $\text{Auth}_\mathcal{P}(d)$ be the set of authoring access control policies that apply to document $d$. Let $\text{Sbj}$ be the set of identifiers of subjects authorized to access documents in $S$. An authoring certificate $ac$ is a tuple $(\text{priv}, \text{sbj}_id, \text{prot}_\text{obj})$, digitally signed by the server, where: priv $\in$ Authoring-privileges, sbj_id $\in$ Sbj; prot_obj is a pair $(\text{id}, \text{at}_\text{el})$, such that $\text{id} \in DR(d)^1$, and at_el is a set of atomic element identifiers belonging to $\text{id}$.

In the following, we denote with the term *valid certificate* an authoring certificate generated according to the policies in $\mathcal{PB}$. More precisely, an authoring certificate $ac=(\text{priv}, \text{sbj}_id, \text{prot}_\text{obj})$ is valid if subject $\text{sbj}$ is authorized to exercise privilege $\text{priv}$ over the set of atomic elements identified by $\text{prot}_\text{obj}$ according to the policies in $\mathcal{PB}$. Moreover, given a subject $s$, we denote with $\text{Cert}(s)$ the set of valid certificates of subject $s$ for the documents in $S$ wrt the policies in $\mathcal{PB}$. The do takes care of sending the certificates to the subjects according to one of the following modes: *on-line; partially on-line; off-line.*

The on-line mode is based on an on-demand method for the certificates generation and distribution. This mode implies the generation of a certificate and its delivery only when it is strictly needed. However this mode has the drawback that the do can become a bottle-neck. The partially on-line mode provides the generation of the authoring certificates needed by the first subject and by all the other subjects that must receive the package as specified in the path conditions generated by the do. Also in this case the do can become a bottle-neck, even if the number of certificate requests addressed to the do is lower. The last mode, the off-line one, provides the preventive generation and distribution of all authoring certificates. Though this strategy could be expensive, it can be executed during the periods in which the working load for the do is lower (e.g. during the night), preventing the do from becoming a bottle-neck.

---

$^1$We recall that $DR(d)$ denotes the set of region identifiers of document $d$. 
Table 4.1: Control data structures for non-modifiable regions

<table>
<thead>
<tr>
<th>Name</th>
<th>Notation</th>
<th>Structure</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control structure for non-modifiable</td>
<td>$NMR_d$</td>
<td>set of $T_{NMR}$, one for each non-modifiable region of $d$</td>
<td>Information used by a subject to verify integrity of non-modifiable regions of $d$</td>
</tr>
<tr>
<td>regions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Control tuple for non-modifiable regions</td>
<td>$T_{NMR}$</td>
<td>$(r_{id}, H_{r_{id}}, NMAE_{d})$</td>
<td>Information corresponding to a specific non-modifiable region $r_{id}$ of $d$</td>
</tr>
<tr>
<td>Control structure for atomic elements</td>
<td>$NMAE_d$</td>
<td>set of $T_{NMAE}$, one for each non-modifiable atomic element of $d$</td>
<td>Control information associated with the atomic elements belonging to a non-modifiable region $r_{id}$ of $d$</td>
</tr>
<tr>
<td>Control tuple for atomic elements</td>
<td>$T_{NMAE}$</td>
<td>$(ae_{id}, position, encrypted-content)$</td>
<td>Information corresponding to a specific atomic element $ae$ belonging to a non-modifiable region $r_{id}$ of a document $d$</td>
</tr>
</tbody>
</table>

The mode is chosen by taking into account the average number of simultaneously active processes, because a high number of processes active at the same time can cause the $do$ to become a bottle-neck.

**Example 4**  Consider three users Alice, Bob and Tom with credentials company management director, manager, and secretary, respectively. Suppose moreover that Bob and Tom work in the R&D department. Consider moreover the policies in Figure 3.4 and information in Table 3.2. Then: (update $\mathbf{c}$. attr, 146, (R1, {&1.Date, &1.Department})) is not a valid certificate, since Alice is not authorized to update attributes of region R1, but only to view their content. By contrast, (update $\mathbf{c}$. attr, 102, (R3, {1.content, 1.ID})) and (update $\mathbf{a}$. attr, 154, (R2, {&4.content, &4.overall descr})) are examples of valid certificates since Tom and Bob are authorized to update the content of node 1 and &4, respectively.

### 4.3.2 Control data structures for document regions

The Control Information Generator module generates different data structures for non-modifiable and modifiable regions. Since non-modifiable regions cannot be altered during the document flow, the control data structure for non-modifiable regions simply contains a hash value for each non-modifiable region. This value is computed by the $do$ before sending the package to the first subject. The idea is that, when a subject wishes to verify the integrity of a non-modifiable region it locally computes the hash value and compares it with the one stored in the data structure. If the two hash values differ, then the region has been modified by a non-authorized subject and
<table>
<thead>
<tr>
<th>Component</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_{id}$</td>
<td>identifier of a non-modifiable region of a document $d$</td>
</tr>
<tr>
<td>$H_{r_{id}}$</td>
<td>hash value computed over the encrypted (with the key of the region) atomic elements belonging to $r_{id}$</td>
</tr>
<tr>
<td>$ae_{id}$</td>
<td>identifier of the atomic element $ae$</td>
</tr>
<tr>
<td>position</td>
<td>it specifies where $ae$’s components are located in the original document $d$ and it is computed by counting, for each component of $ae$, the components that precede it in $d$. This is done by executing a pre-order depth-first left-to-right tree traversal of the graph representation of the document $d$ and assigning a progressive integer number to each atomic element component considered during the traversal. It is important to note that when an element $e$ has some attributes and some children elements the tree traversal assigns, an integer number to the first part of the start-tag of $e$, one to each attribute of $e$, one to the end part of the start-tag of $e$, one to all the atomic element components contained in the children elements, and one to the end-tag of $e$.</td>
</tr>
<tr>
<td>encrypted-content</td>
<td>it contains the encryption of the content associate with the atomic element identified by $ae_{id}$</td>
</tr>
</tbody>
</table>

Table 4.2: Components of the control data structures for non-modifiable regions

Thus the document is corrupted. The control data structure for non-modifiable regions also contains the encryption of the content and the control information associated with the atomic elements belonging to non-modifiable regions. Table 4.1 presents the control data structures for non-modifiable regions of a generic document $d$ in terms of their notation, structure and semantics, whereas Table 4.2 explains the semantics of the components of the control data structures introduced in Table 4.1.

The control information for modifiable regions is more complex than the one for non-modifiable regions because modifiable regions may change dynamically during document flow. Therefore it is not possible to compute only once the hash value for integrity verification. Such a hash value must be recomputed each time the document is modified to allow a subject to verify the correctness of the modifications performed so far on the document. By correctness of the modification we mean that if a subject has modified the document, then it must have the proper authorization. Thus the control information for modifiable regions changes dynamically during the package flow from one subject to another to reflect the operations performed on the document. To make possible the integrity verification of a region the protocol must record information about the last two subjects that have received the package and have an authoring or browsing privilege on that region. More precisely, the control data structure contains, for each region, information on the last two subjects that have confirmed or modified the region, denoted in the following as $s_{\text{last}}$ and $s_{\text{last-1}}$, respectively. We say that a subject $s$ confirms a modifiable region when it verifies the
Table 4.3: Control data structures for modifiable regions

<table>
<thead>
<tr>
<th>Name</th>
<th>Notation</th>
<th>Structure</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control structure for modifiable regions</td>
<td>$MR_d$</td>
<td>set of $T_{MR}$, one for each modifiable region of $d$</td>
<td>Information used by a subject to verify correctness and integrity of modifiable regions</td>
</tr>
</tbody>
</table>
| Control tuple for modifiable regions            | $T_{MR}$ | $(r_{id}, MAE_d, h_{\text{cs}_{last-1}},
|                                              |                       | $h_{\text{cs}_{last-1}} \cdot \text{dig} - \text{sig},
|                                              |                       | $s_{\text{last-1}},
|                                              |                       | $h_{\text{cs}_{last}} \cdot \text{dig} - \text{sig},
|                                              |                       | $s_{\text{last}},
|                                              |                       | h_{\text{serv}_{last-1}}, h_{\text{serv}_{last}}) | Information corresponding to a specified modifiable region $r_{id}$ |
| Control structure for atomic elements           | $MAE_d$  | set of $T_{MAE}$, one for each atomic element belonging to a modifiable region of $d$ | Information used to find portions of a modifiable region $r_{id}$ and to check their integrity |
| Control tuple for atomic elements               | $T_{MAE}$| $(ae_{id}, position, encrypted-content h_{ae, full})$                     | Information corresponding to a specific atomic element $ae$ belonging to a modifiable region $r_{id}$ of $d$ |

integrity of the region, without modifying it, and a subject $s_{\text{last}}$, different from $s$, has modified that region. In particular if a subject $s$ performs a confirmation operation, it establishes that the updates executed by subject $s_{\text{last}}$ are correct wrt to the policies in $\mathcal{P}B$. By contrast, a subject $s$ modifies a modifiable region, when it exercises some authoring privileges over it. Maintaining information concerning the last two subjects is necessary because a subject $s$, before exercising a privilege over a modifiable region, must be able to verify its integrity. To perform this control subject $s$ must know the state of the region when $s_{\text{last}}$−1 has sent the package to the next subject, the set of elements belonging to the region that $s_{\text{last}}$ has modified, grouped by the privilege exercised on them, and information about the authorizations of $s_{\text{last}}$ over that region. All these information are contained in the data structure for modifiable regions.

Before introducing the data structures for modifiable regions, we must introduce an additional information, called cycle path, that the Control Information Generator inserts into the document package. This information is used by the do when the package returns to the do for recovery. It denotes the number of cycles that the package has traversed, till that point. A cycle occurs when a package reaches a subject that has already received it before. Cycle path is used to avoid that a subject, upon receiving a document, inserts in the received version of the document an old version of a document portion.

The control data structures for modifiable regions also contain the encryption of the content and the control information associated with the atomic elements belonging to modifiable re-
<table>
<thead>
<tr>
<th>Component</th>
<th>Sub-component</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_{id}$</td>
<td>-</td>
<td>identifier of a modifiable region of a document $d$</td>
</tr>
<tr>
<td>$h_{\text{id}}s_{last-1}$</td>
<td></td>
<td>hash value computed over the encrypted atomic elements belonging to $r_{id}$ in the document version created by subject $s_{last-1}$</td>
</tr>
<tr>
<td>$h_{\text{id}}s_{last}$</td>
<td></td>
<td>same meaning of hash value computed over atomic elements in the document version created by subject $s_{last}$</td>
</tr>
<tr>
<td>$h_{\text{id}}s_{last-1} \text{dig} - \text{sig}$</td>
<td>-</td>
<td>hash value signed by $s_{last-1}$, calculated over $h_{\text{id}}s_{last-1}$ and $r_{id}$, used to validate $h_{\text{id}}s_{last-1}$</td>
</tr>
<tr>
<td>$s_{last-1}$</td>
<td>-</td>
<td>subject that verified and/or operated over $r_{id}$ just before $s_{last}$</td>
</tr>
<tr>
<td>$h_{\text{id}}s_{last}$</td>
<td></td>
<td>same meaning of $h_{\text{id}}s_{last-1}$, but referred to subject $s_{last}$</td>
</tr>
<tr>
<td>$h_{\text{id}}s_{last} \text{dig} - \text{sig}$</td>
<td>-</td>
<td>same meaning of $h_{\text{id}}s_{last-1} \text{dig} - \text{sig}$, but referred to subject $s_{last}$</td>
</tr>
<tr>
<td>$s_{last}$</td>
<td>-</td>
<td>the last subject that verified and/or operated over $r_{id}$</td>
</tr>
<tr>
<td>$h_{\text{serv}}s_{last-1}$</td>
<td>-</td>
<td>hash value signed by the $do$ calculated over $(h_{\text{id}}s_{last-1} \cup {\text{cycle}_\text{path}<em>{last-1}})$ used to validate $h</em>{\text{id}}s_{last-1}$ after the update of the value of $\text{cycle}_\text{path}$</td>
</tr>
<tr>
<td>$h_{\text{serv}}s_{last}$</td>
<td>-</td>
<td>similar to the component previously illustrated</td>
</tr>
<tr>
<td>$ae_{id}$</td>
<td>-</td>
<td>identifier of the atomic element $ae$</td>
</tr>
<tr>
<td>position</td>
<td>-</td>
<td>it specifies where $ae$’s components are located in the original document $d$</td>
</tr>
<tr>
<td>encrypted-content</td>
<td>-</td>
<td>it contains the encryption of the content associate with the atomic element identified by $ae_{id}$</td>
</tr>
<tr>
<td>$h_{ae}$</td>
<td>-</td>
<td>hash value computed over the encrypted content of the atomic element $ae$</td>
</tr>
<tr>
<td>full</td>
<td>-</td>
<td>hash value computed over $ae_{id}$ and $\text{cycle}_\text{path}$ and signed by the $do$ (if its value is $null$ it means that the atomic element was erased by a previous subject)</td>
</tr>
</tbody>
</table>

Table 4.4: Components of the control data structures for modifiable regions

The control data structures for modifiable regions of a generic document $d$ are introduced in Table 4.3, whereas in Table 4.4 we explain the semantics of the components of the control data structures introduced in Table 4.3.
Among the control information associated with a modifiable region, *certificates* represents a relevant component. This component contains information about exercised privileges and involved atomic elements. More precisely, component *certificates* contains the set of authoring certificates belonging to the subject that has modified the content of the modifiable region, and the set of atomic elements in the considered region actually modified. In particular this component is updated according to the following strategy. When a subject \( s \) wishes to exercise the `delete_elem` privilege it has to insert into *certificates*, for each region that has some atomic elements belonging to the subtree to be deleted, its own certificate for the `delete_elem` privilege on the subtree to be deleted. Moreover if there exists at least one atomic element already deleted by a previous subject in that region, it has to insert in that component the set of atomic elements, belonging to the subtree to be deleted, that have not been yet deleted. This is necessary because, during the execution of the subject protocol, the set of atomic elements that have been deleted by the last subject that has modified a region needs to be exactly determined. When a subject \( s \) exercises an authoring privilege operation different from `delete_elem` over a modifiable region \( r_{id} \), it has to insert in the component *certificates* of the tuple relative to \( r_{id} \) its own certificate for that privilege and the set of atomic elements actually modified.

Other relevant control information are represented by control hash values computed over a given region. This information includes the components: `prev_r`h, `prev_r`digh, `last_r`h, `last_r`digh. Those information are particularly relevant for the integrity verification process and are updated during a *confirmation* or a *modification* operation. A *confirmation* is executed by a subject \( s \) by replacing information in components associated with \( s_{last-1} \) with those in components associated with \( s_{last} \) and by inserting in components associated with \( s_{last} \) information about itself, by setting component *certificates* to `null`, and by leaving unmodified the control hash values. There is a control hash value for each atomic element \( ae \) belonging to a modifiable region of \( d \) calculated over the encryption of \( ae \)'s content and recorded in a component denoted as \( h_{ae} \). During a confirmation the subject that confirms a modifiable region has to re-compute the hash values, contained in components \( h_{ae} \), corresponding to atomic elements that were modified by subject \( s_{last} \), to reflect the new values of the atomic elements. By contrast, a *modification* requires a proper update of the control information by subject \( s \). In particular, if subject \( s \) was the last subject that has previously confirmed or modified that region\(^2\), then it executes a *modification* by inserting in the components associated with \( s_{last} \) information regarding the exercised privileges and involved

\(^2\)We recall that a document can flow back to a subject several times.
atomic elements (component *certificates*) and by updating the control hash values; otherwise, it has to replace information in components associated with $s_{last-1}$ with those in components associated with $s_{last}$, insert in components associated with $s_{last}$ information about itself, the exercised privileges and involved atomic elements (component *certificates*), and the updated control hash values. In this case the values contained in components $h_{ae}$ corresponding to the updated atomic elements are not modified by subject $s$.

Figure 4.2 illustrates a possible path followed by a document $d$ and in particular the value of the most relevant components of the control data structure $MR_d$ corresponding to the region with identifier $r_n$. Subject $S_X$ modifies region $r_n$ by copying information on the $do$ into the components associated with $s_{last-1}$ and by inserting in the components associated with $s_{last}$ its identifier and its valid certificates. Subject $S_Y$ has no access to that region and thus it does not modify it. Subject $S_Z$, after having verified the integrity of the region, modifies it by executing the same procedure performed by subject $S_X$. Finally $S_T$ verifies the region content and confirms it, by copying the information in components corresponding to $s_{last}$ into those corresponding to $s_{last-1}$, and by inserting its identifier in component $s_{last}$.

### 4.3.3 Generalized control information

The *Generalized control information* consists of some information, called *path of document* $d$ and contained in the package, listing the set of subjects that have received the package, and of an hash value, denoted as $H_{NMI}$, computed over a set of information called *non-modifiable information* and signed by the $do$ with its private key. This non-modifiable information can be modified only by the $do$ and corresponds to: *cycle_path*, all control information over non-modifiable regions and atomic elements, the components $(ae_{id}, position, r_{id})$ in all tuples in the control structure for modifiable atomic elements, and the component $r_{id}$ in all the tuples in the control structure for modifiable regions.

The path of document $d$ is used to rebuild as much as possible the path followed by the package, when an error is detected, whereas the hash value is used to check the integrity of the information that are modifiable only by the $do$.

The path of document $d$ is incrementally updated as the package flows from one subject to another. When a subject receives the package, it inserts in the structure a tuple containing its identifier, a counter which keeps track of how many subjects have already received the package, and the identifier of the subject to which it delivers the package. Moreover, the tuple contains an hash
value, signed by the subject with its private key, computed over all the tuples in the structure.

**Definition 3 (Path of document d).** Let \( d \) be an XML document. The Path of document \( d \) (\( \text{Path}_d \)) is a set of tuples \( (s_{id_c}, \text{prog}, s_{id_{next}}, h_{\text{control}}) \), such that: \( s_{id_c} \) is the identifier of a subject, \( \text{prog} \) is the position of subject \( s_{id_c} \) in the path followed by the document, and \( s_{id_{next}} \) is the identifier of the subject to which \( s_{id_c} \) sends the package. Component \( h_{\text{control}} \) is an hash value, signed by \( s_{id_c} \), computed over: \( \{ (t.s_{id_c}, t.s_{id_{next}}, t.prog) \mid t \in \text{Path}_d \land t.prog \leq \text{prog} \} \cup \{ \text{cycle_path}, d_{id} \} \), where \( d_{id} \) is the identifier of \( d \).

Once the Control Information Generator has initialized the above control data structures for an encrypted document \( d^e \), the Dispatcher module generates the package to be sent to the first subject. Figure 4.3 provides a graphic representation of a package \( P_d \).
After the creation of the package $P_d$, the Dispatcher, first of all, locally stores a copy of the package to be used during recovery operations, then signs the package with the private key of $do$, and sends it to the first subject, using the SSL protocol [27].

### 4.4 Subject and Document Originator Protocols

In this section we present the protocols executed by a subject and by the $do$ during a distributed and cooperative update process for an XML document. In particular Section 4.4.1 describes how a subject performs the correctness check for a received package, how it exercises its update rights over the received document and which steps it must follow to send a package to another subject. Section 4.4.1 also shows an example of the subject protocol execution. Section 4.4.2 describes how the $do$ manages a recovery request raised by a subject.

#### 4.4.1 Subject Protocol

A subject $sbj$, according to the chosen key delivery strategy and certificate dissemination mode, obtains from $do$ a set of keys, and a set of corresponding region identifiers, enabling it to decrypt the portions of the document $d$ it is able to access. Additionally, $sbj$ receives from $do$ the set of its authoring certificates, generated according to the access control policies in $PB$, from which it can determine which privileges can be exercised and on which atomic elements of $d$.

The subject protocol consists of three main steps: 1) verification of the package integrity and authentication, 2) package update, and 3) package delivery to the next subject. Section 4.4.1.1 presents an algorithm for performing step 1, whereas Section 4.4.1.2 summarizes the strategies we have devised for steps 2 and 3. Finally Section 4.4.1.3 presents a possible update scenario.
Algorithm 1 An algorithm for verifying package integrity and authenticity

INPUT: The package $P_d$ coming from a subject $s_n$
The receiver subject $sbj$

OUTPUT: A package $P_d$ containing both correct control structures and correct document content

METHOD:

1. Extract from $P_d$ the Package Signature $S_{sn}(P_d)$
2. Let $H_{P_d}$ be an hash value calculated by $sbj$ over $P_d$
3. If $(D_{KU_{sn}}[S_{sn}(P_d)] \neq H_{P_d})$: reject the package
   else: (a) Let $h_{nmi}$ be the hash value calculated by $sbj$ over non-modifiable information in the package $P_d$
   Check the structure $Path_d$.
   If $(h_{nmi} \neq D_{KU_{D}}[H_{NMI}] \lor \text{error in } Path_d)$:
   Send a recovery request to $do$.
   Receive from $do$ another package in which the generalized control information are correct.
   endif
   error := 0
   (b) Let $Reg = \{r_1, \ldots, r_n\}$ be the set of region identifiers belonging to $NMR(d)$ for which $sbj$ has an authorization.
   For each $r \in Reg$:
   Let $h_{reg}$ be the hash value calculated by $sbj$ over $r$'s atomic elements.
   Let $NMR_d[r], H_{r,d}$ be the $H_{r,d}$ component of the tuple belonging to $NMR_d$, with $r_d = d$
   If $(h_{reg} \neq NMR_d[r].H_{r,d})$
   error := 1
   Send a recovery request to $do$.
   Receive from $do$ another package containing correct control structures and correct document content.
   break
   (c) If ($\neg$ error):
   Let $RM = \{r_1, \ldots, r_m\}$ be the set of region identifiers belonging to $MR(d)$ for which $sbj$ has an authorization.
   For each $rm \in RM$:
   (d) If $(MR_d[rm].h_{fig-sig.certificates} = \text{null})$: Control instructions for confirmed regions
   (e) else Control instructions for modified regions
   endif

Figure 4.4: An algorithm for verifying package integrity and authenticity

4.4.1.1 Package integrity and authentication

A subject $sbj$, upon receiving a package, verifies its authenticity and the integrity of the corresponding control information and of the document content it is authorized to access. If no errors occur during this phase, $sbj$ decrypts all the portions of $d$ it is able to access with the received decryption keys.

Then, it can exercise over the decrypted portions all the privileges derived by its authoring certificates. By contrast, if an error is detected, $sbj$ sends a recovery request to $do$ to obtain the last correct version of the package. The steps executed to verify package integrity are presented in the
algorithm in Figure 4.4, whereas Figure 4.5 gives a graphic representation of the main steps of the algorithm.

The overall strategy of the algorithm is to verify the authenticity and the integrity of the received package and of the associated data structures by locally calculating some hash values over the package, by decrypting the hash values contained in the document package $P_d$ with the public key of the subjects that have already received the package and by verifying the correspondence between the locally calculated hash values and the decrypted ones. If these values are different,
the package is considered incorrect and thus is not accepted. A recovery request is raised by the protocol to obtain another package containing the last correct version of the portions detected as corrupted.

The algorithm starts (step 3) with the integrity verification of the package by extracting the package signature and comparing it with a hash value locally computed over the same elements. If the two values are different the algorithm requests the sender subject to send the package again. Otherwise, the algorithm verifies the correctness of the content of non-modifiable information through the comparison of a hash value \( h_{nmi} \) computed over these information with the one stored in the package (step 3.a).\(^3\) The check operated over Path\(_d\) (step 3.a) consists of the verification of the correct correspondence between the subject identifier in a tuple \((s\_id\_next)\) and the corresponding one \((s\_id\_c)\) in the next tuple in the structure. Moreover, all hash values contained in the tuples must be correct, that is, each component \( h_{control} \), decrypted using the public key of the subject specified in the component \( s\_id\_c \), must be equal to that calculated by the algorithm over the information specified in Definition 5. Finally, the last subject specified in Path\(_d\) must be equal to the sender of the received package, to prevent a subject from intentionally not inserting itself in Path\(_d\), with the purpose of inserting old versions of document portions when it receives again the package. The correctness of this structure is important during the execution of the recovery operations, because it is used to rebuild as much as possible the path followed by package \( P_d \) (for more details see Section 4.4.2). If an error occurs the algorithm sends a recovery request to \( do \) to receive a correct version of the package.

Then, the algorithm verifies the integrity of the atomic elements belonging to non-modifiable regions for which \( sbj \) has an authorization (step 3.b) using the same strategy presented above. If an error occurs the algorithm sends a recovery request to \( do \). If no error is detected, the algorithm verifies the integrity of atomic elements belonging to modifiable regions (step 3.c) by verifying, for each modifiable region \( r \) such that \( sbj \) has an authorization on it, the authenticity and the integrity of the information inserted by the last two subjects \( s_{last-1} \) and \( s_{last} \). If the region is confirmed (step 3.d), its content must be correct with respect to the hash value stored in component \( last_r\_h \) and with respect to that in component \( prev_r\_h \). Moreover the hash values in components \( h_{ae} \), referring to the atomic elements belonging to that region, must be correct with respect to the hash values stored in \( last_r\_dig\_h \) and \( prev_r\_dig\_h \). If a region is modified (step 3.e), then by using the hash value in component \( prev_r\_dig\_h \) and the one in the components \( h_{ae} \) of the atomic elements belong-

\(^3\)We denote with D and E the operations of decryption and encryption of the object in square brackets, respectively. Moreover, with \( K_{U_s} \) and \( K_{R_s} \) we denote the public and the private key of a subject \( s \), respectively.


**Figure 4.6: Update and delivery processes**

If an error is detected, a recovery request is sent to the sender to obtain a correct package. Finally, Algorithm 1 returns a package $P_d$, containing both correct control structures and correct document content. A copy of this package is stored for recovery purposes (together with the id of the sender subject, the corresponding value of $prog$ component in $Path_d$ and the current value of $cycle\_path$) by the receiver in its local store.

### 4.4.1.2 Package update and delivery

After a subject $s$ has executed Algorithm 1 on a document package $P_d$, it can read or modify the regions of document $d$ for which it has some authorizations. Then $s$ must update the data structures to keep track of the operations it has performed on document $d$ and send the updated package to the next subject in the collaborative group that satisfies the received path conditions. The operations performed in these steps are graphically summarized in Figure 4.6.
A subject can locally exercise all privileges, authorized by its certificates, apart from privileges insert\_attr and insert\_elem. These privileges must be executed by do, because the corresponding operations could generate new regions. In this case a subject \( s \) must send the package, and new portions it wishes to insert into the document \( d \) to do, that takes care of executing these operations and sending the updated package to the specified receiver. In particular, new inserted atomic elements are marked according to the policies in \( PB \) by do and then inserted in the corresponding new or old region according to their marking. For each newly created region a new entry in the table Key\_info is inserted, whereas control information corresponding to each old region to which new atomic elements have been added is updated to reflect the new content of the region.

If the receiver \( s_r \) of the package is already present in the structure \( Path_d \), the sender instead of sending the package to subject \( s_r \), sends it to the do that updates the value of cycle\_path and of the other involved structures, appends the content of the structure \( Path_d \) in a local store, re-initializes the \( Path_d \) structure and then sends the package to subject \( s_r \). We do such steps to prevent the closure of a cycle in the path of document \( d \). Such an event would allow subject \( s_r \) to replace some portions of \( d \) in the package with old ones, without being detected by any other subject.

### 4.4.1.3 An Illustrative Example

In this section we discuss the example reported in Figures 4.7, 4.8, and 4.9. We focus on the operations executed by some subjects over the atomic elements and control data structures belonging to region R1. In particular that region is composed by three atomic elements with identifiers 1, 3, and 8, respectively. At the beginning of the process the region is covered only by the hash values computed by do, therefore the components in the \( MR_d \) control data structure associated with \( s_{last-1} \) are empty (value null). When \( S_M \) receives the package, region R1 was never modified. This subject possesses two certificates for that region: one containing the update\_attr privilege and the other containing the delete\_attr privilege. First of all the subject checks the integrity of that region by computing two hash values, one over the encrypted contents of the atomic elements belonging to region R1 (\( h1 \)), and the other over the \( h_{ae} \) components associated with those atomic elements (\( h2 \)), and then checks that \( h1 \) and \( h2 \) match the hash values, \( last_r.h \) and \( last_r.dig_h \), associated with R1 and stored in the \( MR_d \) control data structure. Finally, \( S_M \) updates the atomic element 1, deletes the atomic element 8, updates the corresponding control information in \( MR_d \) and sends the updated package to another subject. After a certain number of subjects the package reaches \( S_P \), that can only view the content of region R1. \( S_P \) checks the integrity of R1 by executing
Chapter 4: Cooperative updates in non-colluding byzantine distributed systems

50

The following steps:

- It checks the integrity and authenticity of the inserted certificates \((ac1, ac2)\). Then it determines the set of updated atomic elements belonging to region \(R1\), corresponding to 1 and the set of deleted atomic elements belonging to \(R1\) corresponding to 8. Such sets of atomic elements are obtained by using the information contained in the component certificates of the tuple associated with region \(R1\) belonging to the \(MRd\) control data structure. Obviously if a certificate with privilege different from update_attr, delete_attr or delete_elem is found in the certificates component, an error is raised.

- It locally computes some hash values. Hash value \(h1\) is computed over the \(h_{ae}\) components associated with the atomic elements belonging to \(R1\) not yet deleted or declared as deleted in component certificates associated with \(s_{last}\); \(h2\) is computed over the encrypted content of the atomic element 3, because it has not yet been deleted and it is not declared as modified in component certificates associated with \(s_{last}\); \(h3\) is computed over the encrypted content of all the atomic elements of \(R1\) not yet deleted; \(h4\) is computed over the encrypted content of the updated atomic element 1; \(h5\) is computed over the component \(h_{ae}\).
SP has only a view privilege over R₁.

SP checks the integrity of R₁:
- It checks the integrity and authenticity of the inserted certificates (ac₁, ac₂).
- It locally computes:
  - \( h₁ = H(h_{ae}(1), h_{ae}(3), h_{ae}(8)) \)
  - \( h₂ = H(3) \)
  - \( h₃ = H(1m, 3) \)
  - \( h₄ = H(1m) \)
  - \( h₅ = H(h₄, h_{ae}(3)) \)

- Updated-ae = \{1\}
- Deleted-ae = \{8\}

IF
- \( h₁ = \text{prev}_r\_dig\_h \) AND
- \( h₂ = h_{ae}(3) \) AND
- \{1\} included in ac₁.prot_obj.at_el AND
- \{8\} included in ac₂.prot_obj.at_el AND
- ac₁.prot_obj.r_id = R₁ AND
- ac₂.prot_obj.r_id = R₁ AND
- every ae in deleted-ae = null AND
- h₃ = last_r_h AND
- h₅ = last_r_dig_h

THEN the check is successfully completed.
ELSE error
(in this case there are no errors)

SP confirms R₁.

Figure 4.8: Update flow of a modifiable region (Step 2)

associated with atomic element 3 and h₄, that are the hash values computed over the atomic elements of R₁ not yet deleted.

- It compares hash value \( h₁ \) with the component prev_r_dig_h in MR_d. The correspondence between these two assures that no subject has modified the values associated with the hₐₑ components.

- It compares hash value \( h₂ \) with the component hₐₑ associated with the atomic element 3. Their correspondence and the satisfaction of the previous check assure that the content of the atomic element 3 was not modified.

- It checks that the atomic elements declared as updated or deleted belong to the set of atomic elements contained in the inserted certificates (ac₁, ac₂), and that the region specified in
Chapter 4: Cooperative updates in non-colluding byzantine distributed systems

S Current checks the integrity of R1: it locally computes:

\[ h_1 = H(1m) \]
\[ h_2 = H(h_{ae}(1), h_{ae}(3)) \]

\[
\begin{align*}
\text{IF} & \quad h_1 = \text{prev}_r_h \quad \text{AND} \\
& \quad h_2 = \text{prev}_r_dig_h \quad \text{AND} \\
& \quad \text{prev}_r_h = \text{last}_r_h \quad \text{AND} \\
& \quad \text{prev}_r_dig_h = \text{last}_r_dig_h
\end{align*}
\]
\[
\text{THEN the check is successfully completed} \\
\text{ELSE error (in this case there are no errors)}
\]

---

**LEGEND**

- **ae**: atomic element
- **H(x)**: hash function computed over argument x
- **h_{ae}(y)**: component h_{ae} associated with the ae whose ae_id = y
- **component[pvd]**: value of the data structure "component" in the previous version of document d

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>ae with ae_id = X</td>
<td></td>
<td>encrypted ae of R1</td>
<td></td>
</tr>
<tr>
<td>Xm</td>
<td>modified ae with ae_id = X</td>
<td></td>
<td>encrypted ae of R2</td>
<td></td>
</tr>
<tr>
<td>Xd</td>
<td>deleted ae with ae_id = X</td>
<td></td>
<td>encrypted ae of R3</td>
<td></td>
</tr>
</tbody>
</table>

---

**Figure 4.9: Update flow of a modifiable region (Step 3)**

those certificates is R1, and finally that each atomic element declared as deleted has a null value in its encrypted-content component. These conditions assure that S_M possesses the proper rights to update and delete atomic elements 1 and 8, respectively, and that atomic element 8 was really deleted.

- It compares hash value h3 with the component last_r_h in MR_d. The correspondence between them assures that no subject had modified the content associated with the atomic elements of R1 not yet deleted, after the modification executed by S_M.

- It compares hash value h5 with the component last_r_dig_h in MR_d. The correspondence
between them assures that components \( \text{last}_r \text{.dig}_h \) and \( \text{last}_r \text{-} h \) cover the same content associated with the atomic elements of R1 not yet deleted, in an indirect mode, the former one, and in a direct mode, the latter one.

Before sending the package to the next subject, \( SP \) confirms region R1 by: updating the value of the components \( h_{\text{ae}} \) corresponding to its modified atomic elements 1 and 8; moving the information associated with the components in \( MR_d \) associated with \( s_{\text{last}} \) into those associated with \( s_{\text{last}-1} \); and inserting into the components associated with \( s_{\text{last}} \) the hash values that cover directly \( (\text{last}_r \text{-} h) \) and indirectly \( (\text{last}_r \text{.dig}_h) \) the current content of the atomic elements, together with the identifier of \( SP \). Finally, subject \( SC_{\text{Current}} \) receives the package and checks its integrity. Since region R1 is a confirmed one, \( SC_{\text{Current}} \) must locally compute a hash value \( (h_1) \) over the encrypted content of the atomic elements of R1 not yet deleted and another one over the corresponding \( h_{\text{ae}} \) components. Then, \( SC_{\text{Current}} \) checks that those hash values match both the ones stored in the components associated with \( s_{\text{last}} \) in the \( MR_d \) control data structure and the other ones stored in the components associated with \( s_{\text{last}-1} \). These conditions assure that no subject after \( SP \) has modified the region content and also that \( SP \) itself did not modify the content of the region.

### 4.4.1.4 Correctness of the Subject Protocol

In this section we present correctness results for the subject protocol. The main property that this protocol assures is that a subject is not able to exercise an authoring privilege for which it does not possess the proper authorization. In particular, we show that the protocol can detect whether the document content or the control data structures have been tampered. In what follows we focus on some tampering cases, and we show how the protocol detects those incorrect modifications. Tampering cases on which we focus are the following:

1. modification of non-modifiable information;
2. modification of the document content associated with a non-modifiable region;
3. modifiable region tampering:
   1. use of an authoring privilege for which a subject does not possess the proper authorization;
   2. removal of the control information associated with a modifiable region;
(c) substitution of a modifiable region content and the corresponding control information with those of a previous version;

1) **Modification of non-modifiable information.**

This type of tampering cannot arise because a modification to non-modifiable information implies a corresponding modification of the $H_{NMI}$ hash value, that can be modified only by the $do$, since this hash value is signed with the $do$ private key.

2) **Modification of the document content associated with a non-modifiable region.**

In this case a modification of the content of a non-modifiable region $r_{id}$ implies a corresponding modification of the $H_{r_{id}}$ hash value stored in the tuple of the $NMRd$ control data structure associated with that region. A modification to a $H_{r_{id}}$ hash value implies a corresponding modification of the $H_{NMI}$ hash value. Based on the same reasoning we have used in item 1, we can conclude that this type of tampering cannot arise.

3.a) **Use of an authoring privilege for which a subject does not possess the proper authorization.**

In this case a subject illegally modifies the content of a modifiable region $r_{id}$ without inserting at least one authoring certificate in the component $certificates$ associated with $r_{id}$ or inserting in this component one or more proper and improper authoring certificates or only improper authoring certificates. In the former type of tampering the protocol detects the illegal modification to the content of that region, because at least one of the two hash values stored in components $last_r_{id}$ and $prev_r_{id}$ does not match the one locally computed over the content of $r_{id}$. This is due to the fact that the malicious subject cannot modify both these control hash values according to the new illegal content of the region, because they are signed by two different subjects as required by the protocol. In the latter type of tampering the protocol detects the illegal modification of the content of that region, because at least one of the following checks raises an error. First of all the protocol checks the integrity of the inserted certificates to detect a possible modification of their contents. Then, the protocol checks: 1) whether the subject specified in the authoring certificates matches the one declared in component $s_{last}$, to detect the case in which a subject uses an authoring certificate of another subject to validate its modification; 2) whether the region specified in the authoring certificates matches $r_{id}$; 3) whether the atomic elements not yet deleted and not declared as modified have kept their previous content and that the other ones declared as modified have been modified according to the privileges contained in the inserted certificates.
3.b) Removal of the control information associated with a modifiable region.

This tampering cannot arise because a modification of a modifiable region identifier implies a corresponding modification of the $H_{NMI}$ hash value, that cannot be modified by a subject, as stated in item 1 above. Therefore it is not possible for the subject to delete a whole tuple in $MR_d$. Moreover also assuming that the region content in $d_e$ is empty, there must still be two hash values, i.e. $\text{last}_{r,h}$ and $\text{prev}_{r,h}$, signed by two different subjects, which state that the region was correctly deleted.

3.c) Substitution of a modifiable region content and the corresponding control information with those of a previous version.

In this case a malicious subject $s$ inserts in $d_e$ and in $MR_d$ of its current package version the content associated with the atomic elements belonging to a modifiable region $r_{id}$ and the corresponding values for the components associated with $r_{id}$ in $MR_d$, all stored in a previous version of the package. Since, according to the assumptions given in Section 4.4.1.2, the value of $\text{cycle}_{path}$ and of all involved control data structures are updated whenever a subject finds in the structure $Path_d$ its next chosen receiver, $s$ cannot perform such a substitution, because the control information belonging to a previous version of a package, stored in the local repository of $s$, was necessarily generated on a value of $\text{cycle}_{path}$ that is different by the current one stored in the current package received by $s$ and moreover the current value of $\text{cycle}_{path}$ is not modifiable, as stated in item 1 above.

A type of illegal substitution is however possible, if the following conditions are all satisfied:

1. $s$ keeps a version of the package ($pvp$) that precedes the current one ($cvp$) that $s$ receives;

2. among the subjects that received the package after $s$ only one of them ($s_{bj}$) modified a modifiable region $r_{id}$, exercising only the $\text{update}_{attr}$ privilege over some its atomic elements;

3. the modifiable region $r_{id}$ results not yet confirmed in $cvp$.

According to the previous conditions $s$ is able to copy in $cvp$ the portion of $pvp$ associated with the atomic elements belonging to $r_{id}$, delete the information inserted by $s_{bj}$ in the tuple associated with $r_{id}$ in $MR_d$ of $cvp$ and confirm the new content of $r_{id}$. At this point the region has a previous content and the successor subjects are not able to detect this illegal substitution.

The following approaches presented in this thesis are able to address this last case, at the price however of an increased communication and storage complexity.
Chapter 4: Cooperative updates in non-colluding byzantine distributed systems

4.4.2 Document Originator Protocol

When a subject detects that a package is compromised, it requires from the do a correct version of the document package. In particular, there are two types of recovery requests that a subject may send to the do.

The first type of recovery request is sent when an error to the generalized control information has been detected. In this case the do protocol checks the structure $Path_d$, in the received package attached to the request, and saves the portion of it that is correct. Then, it sends each subject in the cooperative group a message by which it requires the structure $Path_d$ of the package (if any) they have stored, having the value of $cycle_{path}$ equal to the current value stored by the do. By checking (following the criteria explained in Section 4.4.1.1) the received structures $Path_d$, and by matching them with the correct portion of that one in the corrupted package, the do rebuilds as much as possible the path (denoted in the following as $rebuilt-path$) followed by the package. This is executed by considering the structures $Path_d$, obtained from the subjects, in ascending order with respect to the number of subjects they contain. The path saved by the do is matched with the first structure $Path_d$. Every match generates a partially $rebuilt-path$ which is used in the subsequent matches. Such path consists of the subjects which appear in both paths and of the subjects contained in the path with the highest number of subjects. The rebuilding process terminates when there are no more paths to evaluate or when two paths have, in a tuple, equal value for component $prog$ and different values in one of the other components. Now it is possible to find the last correct version of the package by requiring the subjects listed in the $rebuilt-path$ (beginning from the one with the highest value in the component $prog$ and stopping the requesting process when a correct version is found) their last sent and stored package having the value of $cycle_{path}$ equal to that one stored by the do.

Finally, the do appends the $rebuilt-path$ to the path (denoted as $global-path$) already saved in a local store, re-initializes the $Path_d$ structure, updates the value of $cycle_{path}$ as well as the data structures of the received package and sends this updated package to the subject that has required the recovery.

The second type of recovery request is sent when an error in the content of a region is discovered. In this case if the error affects only non-modifiable regions, the do can directly solve the problem without the help of other subjects, because it has stored the original value of this information in its local repository. It can thus replace the corrupted information with the saved ones. Otherwise, if the $Path_d$ structure has been compromised the do rebuilds the last portion of the path followed by the package, by using the strategies explained above. In any case the do, by using the information
in table Key_Info, selects the subjects from which it requires a package containing the last correct version of the corrupted modifiable regions. The request process is executed starting from the selected subject with the highest value in component \( prog \) in the global-path obtained appending the rebuilt-path to the one already saved and stopping the process when a correct version of the region is found or the selected subject has a value in component \( prog \) less than the one of the subject that has received the last correct version of the region found and saved by the do during a previous recovery session. If a correct version of the region is not found through this process the initial version of the region is inserted into the package. Finally, the do re-initialize the Path_k structure, updates the value of cycle_path as well as the data structures of the received package and sends this updated package to the subject that has required the recovery.

### 4.5 Complexity Analysis of the Proposed Approach

In this section we present a complexity/cost analysis for the most relevant operations executed by the protocols on which our approach relies and we compare them with the equivalent operations executed in a conventional centralized system. In particular we evaluate the following complexity/cost measures:

1. communication cost;

2. size of exchanged information;

3. number of certificates generated for a document \( d \);

4. execution time of the integrity check protocol and distributed view generation/decryption vs execution time of the centralized view generation/encryption and distributed decryption.

The complexity/cost is expressed in terms of several parameters that are reported in Table 4.5. According to the context such parameters will be interpreted as sets of elements or as the cardinalities corresponding to these sets of elements.

Before evaluating the above-mentioned complexity/cost measures we believe that an explanation of what a conventional centralized system means is required to better understand the comparison between our distributed approach and a centralized one.

A conventional centralized system, that manages collaborative updates of an XML document, has to generate the document view for each subject involved in the collaborative process,
Table 4.5: Parameters involved in the complexity/cost analysis

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{P}$</td>
<td>the set of policies that apply to a specified XML document $d$</td>
</tr>
<tr>
<td>DocAE($d$)</td>
<td>the set of atomic elements associated with $d$</td>
</tr>
<tr>
<td>DR($d$)</td>
<td>the set of regions associated with $d$</td>
</tr>
<tr>
<td>CG</td>
<td>the collaborative group</td>
</tr>
<tr>
<td>Path$_d$</td>
<td>the path of document $d$</td>
</tr>
<tr>
<td>$N_i$</td>
<td>number of interactions, that is, the number of sessions per document opened by the subjects in a centralized approach (a subject can open more than one non-simultaneous session per document) or the number of subjects reached by the package in a distributed and cooperative approach (a subject reached more than one time by the same package is counted exactly as many times as it is reached)</td>
</tr>
</tbody>
</table>

4.5.1 Communication cost

The communication cost is estimated in term of the number of messages exchanged among the $do$ and the various subjects. We estimate such cost for both a conventional centralized
system and our distributed approach. We are thus able to compare those two approaches.

1. In a conventional centralized system the number of messages sent by the do and the subjects is equal to two times the average number of access requests ($AR_{avg}$) multiplied by the number of interactions, that in this case corresponds to the number of sessions opened to issue access control requests. The resulting communication cost is thus: $2N_i \cdot AR_{avg}$.

2. In our distributed approach the number of messages sent by the do or by the subjects involved in the collaborative update process, is as follows:

\[
\text{\#sent-msg} = (N_i + 1) + 2(R_R + P_R) + 2P_R(CG - 2) + 2SRM(R_R, \min\{P_R, (N_i - 2)\})
\]

where:

(a) $R_R$, with $0 \leq R_R \leq (N_i - 1)$, is the number of region recovery requests. The upper bound for this parameter is $(N_i - 1)$, because the first subject receiving the package from the do does not certainly need a recovery.

(b) $P_R$, with $0 \leq P_R \leq (N_i - 1)$, is the number of path recovery requests. Also in this case the upper bound for this parameter is $(N_i - 1)$, because the first subject receiving the package from the do does not certainly need a path recovery.

(c) $(N_i + 1)$ is the number of messages, containing the package, needed to reach, one or more times, all the interacting subjects and to return to the do.

(d) $2(R_R + P_R)$ is the global number of recovery requests and answers

(e) $2P_R(CG - 2)$ is the total number of path recovery requests and answers sent to/by the subjects during a cooperative update process. Whenever a path recovery request reaches the do, the do sends all the subjects in the cooperative group, apart the path recovery request sender and the subject from which that subject has received the corrupted path, a request to obtain the last path they have received within the package.

(f) $2SRM(R_R, \min\{P_R, (N_i - 2)\})$ is the number of sent recovery messages needed to manage $R_R$ region recovery requests, given $P_R$ path recovery requests. In particular, we consider the worst case in which both the region recovery requests and the path recovery requests are respectively sent by the last $R_R$ and $P_R$ subjects in the path. Function $SRM(x, y)$ is defined as follows:

\[
SRM(x, y) = \sum_{j=N_i-2-(y-1)}^{N_i-2} \min\{j, N_i - 2 - y, CG\}, y \leq N_i - 2
\]
Note that values for variable $y$ must be less or equal than $N_i - 2$, because when the number of path recovery requests is equal to $N_i - 1$ the recovery protocol does not send any region recovery message, thus the behavior followed is the same of when $y = N_i - 2$. Moreover, for each region recovery request the do sends a number of messages, to obtain the correct version of one or more regions, equal to the number of subjects that precede the last one in the Path$_d$, and however at most $CG$ messages.

**Proposition 1 (Communication cost).** Let $r_r$ and $p_r$ be rational numbers such that $r_r = \frac{R_r}{N_i}$ and $p_r = \frac{P_r}{N_i}$. Our distributed approach has a lower communication cost than a conventional centralized system when $AR_{avg} \geq (1 + p_r + r_r + p_r \cdot CG + r_r \cdot CG)$. According to the above results it is clear that whether our approach is more efficient than a centralized one depends on the frequency of recovery. Based on this observation we plan to extend our system with an adaptive behaviour for recovery management. The adaptive behaviour will allow the system to use a centralized or distributed protocol according to an estimated average number of access requests and the regions/path recovery rates.

### 4.5.2 Size of exchanged information

Here we are interested in the amount of data exchanged in each step of a collaborative update process in both a conventional centralized system and in our distributed approach. Tables 4.6 and 4.7 contain data useful for such estimation in both approaches. More precisely, in Table 4.6 we specify the size of the building blocks that compose a package for the centralized approach, the distributed one, and for an access request.

By contrast, in Table 4.7 we show the size of three basic package structures that compose a package for the distributed approach: a **modifiable atomic element**, a **modifiable region**, and a **path specification**, in terms of the size of their building blocks.

1. In a *conventional centralized system* the worst case occurs when the view to be generated for the next receiver is equal to the whole XML document $d$. Indeed the system has to generate a symmetric session key and encrypt the whole XML document before sending it to the next receiver. Here we do not consider the generation and distribution of the corresponding authoring certificates, that will be treated in Section 4.5.3

\footnote{Note that also in this case we consider the worst case in which all the considered subjects are distinct.}
Table 4.6: Size of the building blocks of a package/access request

<table>
<thead>
<tr>
<th>Building Block</th>
<th>Size</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>$S(id)$</td>
<td>size of a region/atomic element/subject identifier</td>
</tr>
<tr>
<td>hash</td>
<td>$S(hash)$</td>
<td>size of a hash value</td>
</tr>
<tr>
<td>digital signature</td>
<td>$S(digital signature)$</td>
<td>size of a digital signature</td>
</tr>
<tr>
<td>$d'$</td>
<td>$S(d')$</td>
<td>size of the encrypted document</td>
</tr>
<tr>
<td>ar</td>
<td>$S(ar)$</td>
<td>size of the access request declaration given in terms of atomic elements to be updated</td>
</tr>
<tr>
<td>ac</td>
<td>$S(ac)$</td>
<td>size of an authoring certificate</td>
</tr>
<tr>
<td>$up_{ae}$</td>
<td>$S(up_{ae})$</td>
<td>size of the updated portion ($ap$), sent to replace the old portions of an atomic element ($ae$), with old size $S(\text{ae})$, contained in the document.</td>
</tr>
</tbody>
</table>

signature to the view itself, forming a package to be sent to the designated receiver. Moreover, during a step of a centralized collaborative update process a subject $s$ in the worst case, that is when a different access control policy applies to each atomic element and all these policies apply to $s$, sends a number of access requests, and corresponding updated portions, equal to $AE(d)$, the number of atomic elements that compose $d$. We can estimate the amount of data exchanged between the centralized system and a subject during a step of the centralized collaborative update process as follows.

**Proposition 2 (Size of exchanged information for the centralized approach).** Let $d$ be an XML document, $CP_d$ be the generated corresponding package for the centralized approach, $S(CP_d)$ be the size of $CP_d$, and $S(AR)$ be the size of $AR$, where $AR$ is the set of access requests sent by a subject $s$ during a step of a centralized collaborative update process. In the worst case: $S(CP_d) + S(AR) \leq c \cdot AE(d)$, $c \in \mathbb{N}$.

2. In our distributed approach the worst case occurs when the following conditions hold: 1) the number of regions associated with $d$, $R(d)$, is equal to the number of atomic elements of $d$, $AE(d)$; 2) all the regions are modifiable ones; and 3) the current subject updates all the atomic elements. The size of the information exchanged during a step of a distributed collaborative update process is equal to the size of the package sent by the current subject to the next chosen receiver. The size of such a package is defined as follows.

**Proposition 3 (Size of a package for the distributed approach).** Let $d$ be an XML document, $DP_d$ be the generated corresponding package for the distributed approach, and $S(DP_d)$
be the size of \( DP_d \). In the worst case: \( S(DP_d) \leq c \cdot AE(d), \ c \in \mathbb{N} \).

According to the above results, it is clear that the size of exchanged information in both approaches linearly grows wrt the number of atomic elements that compose the original document, thus our distributed approach can offer a bandwidth cost similar to the one for the centralized approach. The major benefit of our approach wrt the centralized one remains the reduced number of messages sent during the collaborative update process, thus the communication cost is still the effective parameter of choice between the distributed approach and the centralized one.

### 4.5.3 Number of certificates generated for a document \( d \)

The worst case occurs when the graph representation of the document \( d \) is a list of nodes and all the policies that apply to this document contain the `delete_element` privilege with propa-
Chapter 4: Cooperative updates in non-colluding byzantine distributed systems

63

gation option equal to CASCADE. In particular, the policies apply to the document as follows: the first policy applies to the root of the document, the second policy applies to the child (second node in the list) of the document and so forth. According to this scenario the number of certificates that must be generated is equal to \( \sum_{i=0}^{P-1} S_i + (AE(d) - i) \), where \( S_j \) is the set of subjects satisfying the \( j^{th} \) policy \((1 \leq j \leq P)\). The number of certificates is always less than or equal to \( max\{S_j | 1 \leq j \leq P\} \cdot P \cdot (AE(d) - \frac{P-1}{2}) \). In the worst case \( P \) is equal to \( AE(d) \) and thus the number of certificates has an upper bound equal to \( max\{S_j | 1 \leq j \leq P\} \cdot \frac{AE(d)^2}{2} \).

Only the distributed approach presents this cost associated with the generation and dissemination of the certificates. Moreover, the possible update and/or revocation of subject credentials and access control policies add a further cost due to the generation of new certificates and the revocation of the ones no more valid. Since our distributed approach postpones these events at the end of a collaborative update process, this last cost can be managed off-line without impacting on the process itself. According to the result above to make our approach better than the centralized one we propose an adaptive system that evaluates the number of certificates to be generated before starting a collaborative process to choose the best strategy of generation and dissemination of certificates among those presented in Section 4.3.1.

4.5.4 Execution time of the integrity check protocol and distributed view generation/decryption vs execution time of the centralized view generation/encryption and distributed decryption

In this section we analyze and then we compare the time cost required to enable the next receiver to view document portions for which it possesses a privilege, and to modify the document content according to its modification rights stated in the policies belonging to the \( PB \). In a conventional centralized system this time is spent by the centralized system to generate and encrypt the receiver document view and by the receiver to decrypt such a view, whereas in our distributed approach this time concerns the execution of the integrity check protocol, and the generation/decryption of the receiver view.

1. In a conventional centralized system the worst case occurs when a different access control policy applies to each atomic element of the document and also all these policies apply to the next receiver, implying that the view to be generated and encrypted by the centralized system and decrypted by the receiver consists of the whole document. The time required to accomplish the task of generating and encrypting the next receiver document view by the
Proposition 4 (Centralized view generation/encryption and distributed view decryption time). Let \( d \) be an XML document, \( P \) be the set of access control policies that apply to \( d \), \( nr \) be the next receiver, and \( T(\text{view}) \) be the time required by the centralized system to generate and encrypt the \( nr \)'s document view and by \( nr \) to decrypt that view. In the worst case: \( \exists c \in \mathbb{N}: T(\text{view}) \leq c \cdot (AE(d))^2 \).

2. In our distributed approach the worst case occurs when: the number of regions \( R(d) \) is equal to the number of the atomic elements \( AE(d) \) that compose the document \( d \); and there is at least a policy with privilege \( \text{update\_attr} \) that applies to each region. This is the case in which the protocol requires the highest number of operations to perform the region integrity check and the number of regions that requires such a set of operations is maximum. Moreover the receiver view to be generated and decrypted consists of the whole document. In this case the integrity check of an updatable region \( r \) requires the following steps:

   (a) Integrity check of the information inserted by the last but one subject \( (s_{last-1}) \) in the region \( r \), that requires the local computation of an hash value and the decryption of a digital signature (component \( h_{c_{\text{last-1}}} \text{dig-sig} \)).

   (b) Integrity check of the components \( h_{\text{ae}} \) associated with the atomic elements belonging to \( r \), that requires the local computation of an hash value and its comparison with the component \( \text{prev}_{r\_\text{dig\_h}} \).

   (c) Local computation of an hash value, one for each atomic element belonging to \( r \), over the encrypted content of an atomic element.

   (d) Integrity check of the information inserted by the last subject \( (s_{last}) \) in the region \( r \), that requires the local computation of an hash value and the decryption of a digital signature (component \( h_{c_{\text{last}}} \text{dig-sig} \)).

   (e) Check that the value contained in component \( \text{last}_{r\_\text{dig\_h}} \) is equal to the hash value locally computed over the hash values computed over the atomic elements belonging to \( r \).

   (f) Check that the value contained in component \( \text{last}_{r\_h} \) is equal to the hash value locally computed over the atomic elements belonging to \( r \).
Table 4.8: Time required to perform the basic integrity check protocol operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time Notation</th>
<th>Time expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash</td>
<td>$T(\text{hash})$</td>
<td>time required to compute an hash value</td>
</tr>
<tr>
<td>digital signature</td>
<td>$T(\text{digital signature})$</td>
<td>time required to decrypt a digital signature using the corresponding public key</td>
</tr>
</tbody>
</table>

Table 4.9: Time required to perform the integrity check protocol operations

(g) Integrity check of the certificate inserted in the component certificates by the last subject ($s_{\text{last}}$), that requires the local computation of an hash value and the decryption of a digital signature, only if the region contains an atomic element that is an attribute, since only in this case a certificate is inserted and a correct update over that attribute can be performed by the last subject.

Table 4.8 shows the time required to perform some basic operations, whereas Table 4.9 shows the time required to perform the integrity check protocol operations.

Furthermore given the set of regions accessible by the receiver, denoted as $\text{AccReg}$, the view generation process requires a sequential research in the package to find out each accessible region and their atomic elements.

**Proposition 5 (Integrity check protocol and distributed view generation/decrption time).**

Let $d$ be an XML document, $P_d$ be the corresponding document package, $nr$ be the next receiver, and $T(\text{view})$ be the time spent to check the package integrity and to generate/decrypt the $nr$’s document view. In the worst case: $\exists c \in \mathbb{N}: T(\text{view}) \leq c \cdot \text{AE}(d)^2$. 
The above results shows that both approaches require a similar time cost to enable a receiver to have access to its document view. The communication and generation/dissemination costs are the parameters according to which it is possible to choose the best approach.
Chapter 5

Cooperative updates in colluding byzantine distributed systems

Contents

5.1 Introduction ............................................ 68
5.2 Preliminary definitions ............................... 70
5.3 Control information .................................. 71
5.4 Protocol for managing distributed updates ............. 76
  5.4.1 Path and deltas integrity check ...................... 78
  5.4.2 Document integrity check .......................... 80
  5.4.3 Protocol for document and path modification .......... 81
5.5 A running example .................................... 84
5.6 Correctness proof ..................................... 85

Figures

  5.1 Document pre-processing ............................. 69
  5.2 An example of document path ........................ 72
  5.3 Document package structure .......................... 74
  5.4 Example of data structures within a document package ....... 75
  5.5 Example of document path extension ................... 82
  5.6 Effective path checklist data structures, when the security level is set to 2 .... 83
  5.7 Example of collusion detection using the effective path checklists ............. 84

67
5.1 Introduction

In this chapter we present an approach for the distributed and cooperative updates of XML documents, supported by our framework, particularly suited for colluding byzantine distributed systems. The proposed approach is based on a specific infrastructure, that specializes the one already presented in Chapter 3. The architecture we have designed, which is illustrated in Figure 5.1, is based on the following principles. When a document must be updated and accessed by several subjects it undergoes an encryption phase, in which all the document portions to which the same access control policies apply are encrypted with the same symmetric key, using a symmetric cipher system. Then, the encrypted document is sent to the first subject belonging to the \( CG \). Each subject in the \( CG \) only receives the symmetric key(s) for the portion(s) it is enabled to see and/or modify. Additionally, attached to the encrypted document, a subject receives some control information, with the purpose of making it able to locally verify the correctness of the updates performed so far on the document, without the need of interacting with the do. The encrypted document and the control information form the document package. In the previous chapter each subject, under specific assumptions, once receiving the document package, is able to verify, without interacting with the do, whether the operations performed till that point on the document are correct. The corresponding infrastructure relies on the assumption that each time a subject detects that a document portion is in an inconsistent state (that is, a previous subject has operated on that portion violating the stated access control policies) sends a recovery request to the do to obtain the last correct version of the package. This means that the developed protocols are not resistant to the collusion of two or more subjects such that one masquerades the unauthorized modifications of the other. This is due to the possibility for a subject of certifying the correctness of the operations executed over the document by a subject preceding the successor subjects, that, on the basis of that certification, will assume the correct execution of those operations.

The current approach extends our previous one by relaxing this assumption. In this chapter, we focus on the update privilege only, which is the most relevant and widely used authoring privilege in our model. We believe that this is a major extension because it makes our infrastructure suitable for a larger variety of domains, in which the possible collusion of two or more subjects must be taken into account. According to the infrastructure proposed in this chapter, the path that the document must follow is partially predefined by the do before sending the package to the first subject in the \( CG \). However, the subjects receiving the package can dynamically reconfigure the path by extending it during the document flow. This is an important feature since it is often the case that the
document path must be extended due to unexpected events, or because a subject chooses to send the
document to a group of subjects that execute some specialized operations over the document that
it is not able to perform (e.g., a top executive manager delegates the update of the conditions of a
sale contract to the sale manager and to the legal office). Problems due to subjects collusion are
prevented in this approach by recording in the document package a certified history of the modifica-
tions performed on the associated XML document. The updates inserted in the history are encrypted
using the same strategy that drives the encryption of the document. This ensures that a subject can
see only the modifications associated with portions it is authorized to access. Each subject updating
the document adds an entry to this history, as well as a set of control information. Both the his-
tory and the control information are enclosed in the document package. Each subject receiving the
document is able to locally verify that all the updates performed so far on the document have been
made in accordance to the stated policies. Moreover, each subject can build its current view of the
document content, according to the stated access control policies, using the keys received by the do.
When the document will return to the origin host it will have gathered all the information necessary
to reconstruct the intended updates by the users.

The remainder of this chapter is organized as follows. Section 5.2 introduces some pre-
liminary concepts and definitions that are necessary to explain the protocols proposed in this chapter.
Section 5.3 presents the control information required to support distributed document updates and
the extensions of the predefined path, whereas Section 5.4 describes the protocols for checking the
Chapter 5: Cooperative updates in colluding byzantine distributed systems

integrity of the document content and of the path that the document must follow, and for building the current document view of a subject wrt the access control policies that apply to that document. Section 5.5 presents an example illustrating the robustness of the proposed protocol, whereas Section 5.6 gives an informal proof of correctness for the proposed protocol.

5.2 Preliminary definitions

In this section, we present the basic concepts and definitions on which the proposed protocol relies. In particular, we introduce the notion of certificate, which is used to prove that a subject has the proper rights to execute an update operation over a document.

In this chapter, we focus only on policies containing one of the browsing privileges or the update privilege. According to this assumption document regions can be classified into modifiable or non-modifiable regions. A modifiable document region is a region consisting of atomic elements to which at least one access control policy containing the update privilege applies. This region is called modifiable because its content can be modified during the document flow. By contrast, a non-modifiable document region is a region consisting of atomic elements to which only access control policies containing a browsing privilege apply. The content of a non-modifiable document region can be only read by authorized subjects and no subjects can modify it. For each modifiable document region the Control Information Generator module (CIG), reported in Figure 5.1, generates a set of update certificates, according to the access control policies stored in the PB. The CIG generates a certificate for each subject that satisfies at least a policy containing the update privilege, that applies to that region, with the following structure.

**Definition 4 (Update certificate)** An update certificate uc is a tuple: (update\_attr, subj\_id, prot\_obj), digitally signed by the do, where:

- **subj\_id** is the id of the subject that can exercise the update\_attr privilege over the protection object prot\_id.
- **prot\_id** is a pair (r\_id, at\_el) such that r\_id ∈ DR(d), and at\_el is a set of atomic element identifiers, belonging to the region r\_id.

The do signs each generated certificate with its private key for authentication purpose and takes care of sending all the certificates to the corresponding subjects. Update certificates are used by a subject, that modifies a document portion, to prove its right to modify that document portion.
to the subsequent receivers of the document. Therefore, whenever a subject modifies a document (or a document portion), it has to add the proper valid certificate to the document control structure presented in the next section. In the following, with the term valid certificate we refer to a certificate generated according to the policies in \( PB \).

### 5.3 Control information

In this section we introduce the control information, attached to the encrypted document, used by subjects to check integrity of the document content.

Next definitions introduce two basic concepts of our approach: the notion of path of a document, representing an ordered list of subjects to which the document has to be sent; and the notion of document deltas, an incremental representation of the modifications performed on both the document and the path by the various subjects belonging to the document path.

**Definition 5 (Path of a document \( d \))** Let \( d \) be an XML document. The path of a document \( d \), denoted as \( \text{Path}_d \), is an ordered list of subject identifiers, \( <s_1, s_2, \ldots, s_n> \), such that the \( i^{th} \) element of the list, denoted as \( s_i \) (\( i=2,\ldots,n \)), is the identifier of the \( i^{th} \) intended receiver of the document \( d \), whereas \( s_1 \) denotes the identifier of \( do \).

The path of a document \( d \) is included in the document package and it is used for checking the consistency of the modifications operated by subjects. Such data structure is generated by CIG and signed with the private key of \( do \), during the pre-processing phase of the protocol.

**Example 5** Suppose that the generated path for a document \( d \) is the one illustrated in Figure 5.2. In this case, the CIG generates the list of subjects ids, associated with the corresponding credentials (inserted within the rectangles in the figure) and signs it with the private key of \( do \). Note that the id of the \( do \) is always the first in \( \text{Path}_d \). Therefore, the signed path of \( d \) is the following: \( S_{do} (<do, 102, 104, 154>) \).

Whenever a subject executes an update of some portions of the document, it has to insert in the package some information describing which are the modified elements, grouped by the regions they belong to, and the proper valid certificates that guarantee the correctness of those operations. In particular, the approach used for obtaining the last version of a given document \( d \) is to give subjects access to the portions of the original document \( d \), for which they have an authorization,
and to enable them to locally compute all new values of those portions of $d$. These information are collected into a data structure, called *region update record*, defined as follows.

**Definition 6 (Region update record)** A region update record, denoted as $RUR$, is composed of three fields: (region, UC, AEUL), where region is the id of a modifiable region, UC is a valid update certificate for that region, and AEUL is a set of updates over the atomic elements belonging to that region. Each element $(\text{at_el, pos, } E_k(\text{new_value}))$ of AEUL, is an atomic modification over a single element of a document region where: at_el is the identifier of the atomic element to modify, pos is the position of the atomic element within the document, expressed by an XPath expression, and $E_k(\text{new_value})$ is the new value of the atomic element encrypted with the key associated with the corresponding region.

All the modification operations executed by a subject to the document or to the path are inserted into a data structure, called *Atomic delta*, formally defined as follows.

**Definition 7 (Atomic delta)** An atomic delta is a two-fields structure: $<$op_type, op_arg$>$. The field op_type can have one of the following values: NO_OP, OP_UPDATE or OP_EXTENSION. The meaning of those values is as follows:

NO_OP is the null operation type. No additional data is needed, so the corresponding op_arg field has a null value.

OP_UPDATE is used when a subject wants to modify a document portion. The corresponding op_arg contains a set ({$RUR_i, RUR_j, \cdots$}) of region update records, one for each region of the document to which at least an atomic element modified by the subject belongs to.
OP_EXTENSION is used when the subject wants to operate an extension to the original document route Path_d. The op_arg field consists of an ordered list < s_1, s_2, ⋯, s_m >, where the i-th element of the list is the identifier of the i-th intended receiver of the document wrt the specified extension.

All the operations executed by the subjects that have received the document are stored in a data structure called Deltas_d, that we define formally as follows.

Definition 8 (Deltas of a document d) Let d be an XML document. The deltas of a document d, denoted as Deltas_d, is a list of operations < delta_{s_1}, delta_{s_2}, ⋯, delta_{s_n} >. Let s_i (i=1, ⋯, n) be the i-th receiver of the document, then delta_{s_i} consists of: a) an atomic delta containing a NO_OP operation, if s_i has not executed any modification; or b) an atomic delta containing an OP_UPDATE operation, if s_i has only updated some atomic elements of the document; or c) an atomic delta containing an OP_EXTENSION operation, if s_i has only extended the path; or d) two atomic deltas containing: an OP_UPDATE, and an OP_EXTENSION operation, respectively, if s_i has updated some atomic elements of the document and it has also extended the path. No other content for delta_{s_i} is correct.

The last definition imposes that, even if a subject s does not execute any modification operation, it has to add an atomic delta containing a NO_OP operation in its delta_s, inserted in the Deltas_d data structure. This is required to check that the path was correctly followed, that is, no subject was deleted by the path.

The Encryption Module (EM) and the CIG respectively sign, with the private key of do, the encrypted document d' and the original path, Path_d, in order to authenticate their contents, which will not be modified anymore. In order to certify the integrity of all the deltas inserted in the Deltas_d data structure, each receiver s must sign its delta_s together with the other ones already stored in the Deltas_d data structure.

The next definition presents the data structure in which these signatures are stored, named Signatures.

Definition 9 (Signatures) The Signatures data structure is a tuple (Signed_path, Signed_doc, Signed_deltas) where: i) Signed_path is the signature computed over Path_d, at the beginning of the update process by CIG, with the private key of do denoted also as S_{do}(Path_d); ii) Signed_doc is the signature of the encrypted document d, computed at the beginning of the update process by
Figure 5.3: Document package structure

$EM$ with the private key of $do$, denoted also with $S_{do}(d')$; iii) Signed deltas is a list of signed values $<S_{s_1}(v_1), \ldots, S_{s_n}(v_n)>$, where $v_i (i=1, \ldots, n)$ is the list $<\text{delta}_{s_1}, \ldots, \text{delta}_{s_n}>$.

Figure 5.3 shows the structure of the whole document package.

**Example 6** Consider a cooperative group formed by subjects \{secretary sid = "102", accountant sid = "104", company management director sid = "146", manager sid = "154", notary sid = "138"\}, and suppose that only 102, 104, and 154 take part in the update process. Figure 5.4 shows the data structures contained in the document package just after the modifications performed by 154. In the example, 154 has added an extension (subjects 138 and 146) to the original path, with an OP_EXTENSION operation. Moreover, 154 has modified some atomic elements ($a_{e_1}, a_{e_2}, a_{e_3}$, and $a_{e_4}$) that belong to two different document regions ($R_1$ and $R_2$). Thus, 154 has added an OP_UPDATE type operation with two region update records, one for $R_1$ (that includes the modifications to $a_{e_1}$ and $a_{e_2}$) and one for $R_2$ (that includes the modifications to $a_{e_3}$ and $a_{e_4}$). In the example, the key for the encryption of region $R_1$ is $k_1$, whereas the key for the encryption of region $R_2$ is $k_2$. Before sending the document to the next subject in the path (this subject is not do but 138 because of the path extension), 154 signs all the deltas in the $\text{Deltas}_d$ data structure, and inserts this signature in the $\text{Signatures}$ data structure.

To avoid the collusion of two or more subjects we impose that subjects, whenever they receive a package, notify their position within the path associated with the received XML document to a subset of the subjects that belong to the cooperative group. The protocol is parametric in that the number of subjects that must be notified depends on the number of subjects that are supposed
to collude. The notifications avoid the possibility of deleting a subject belonging to an extension of the path, because there exists at least one honest subject (in the worst case the do) that can see this incorrect operation. Each subject in the cooperative group locally stores in a particular data structure, named Effective path checklist, all the notifications, grouped by the document they are associated with, that it received from the other subjects in the group and its position in the path associated with a received document, and uses this information when checking the validity of the path. This data structure is formally defined as follows.

**Definition 10 (Effective path checklist)** Let s be a subject belonging to the cooperative group. The effective path checklist for s is a data structure locally managed by s that consists of a set of pairs

---

1 More details about the number of subjects to which the notification must be sent and the method adopted to choose this set of subjects are explained in Section 5.4.3.
(si, position) where: a) si is the subject that notifies s of the reception of an XML document, or s itself, whenever it receives an XML document; b) position is the position occupied by subject si in the path associated with that XML document.

5.4 Protocol for managing distributed updates

Before sending the document d to the first subject, the EM selectively encrypts the document in accordance to the strategy defined in Chapter 3 and recalled in Section 5.2. The result of the encryption phase is:

1. The encrypted document $d'$;

2. A table, named Key-info, that maps each region to the key used to encrypt it. More precisely, the table contains an entry for each document region, reporting information about the set of atomic elements that belong to that region, the key used to encrypt those atomic elements, and the set of policies that mark those elements. This table is locally stored and managed by the do.

Moreover, the EM signs the encrypted document $d'$, with the private key of do, in order to authenticate its content, which will not be modified anymore. Before sending the document the CIG generates the data structures that are used to track the modifications of the original document, to detect unauthorized modifications to the document and to access the encrypted data in accordance with the stated access control policies, and inserts them in a package together with the encrypted document $d'$. The whole process of document pre-processing is illustrated in Figure 5.1 and is composed by three steps.

The first step initializes the $Deltas_d$ data structure. This data structure is initialized with a $NO_OP$ type operation, added by the CIG and it is filled by subjects during the update process. The approach proposed here is to not allow any modification to the original content but, instead, to keep track of every operation performed over the document ensuring that:

1. Each $OP\_UPDATE$ operation executed over the document and inserted in the $Deltas_d$ data structure by a subject is authorized by the access control policies in $PB$.

2. Each updated value of an atomic element of the document $d$, generated by an $OP\_UPDATE$ operation and registered in the $Deltas_d$ data structure, is accessible only by authorized subjects, according to the stated access control policies.
3. The list of operations is protected against tampering from malicious subjects.

The second step initializes the $Path_d$ data structure, that is, the intended route the document is supposed to follow during the modification process. The $Path_d$ data structure is signed by $CIG$, using the private key of $do$. Subjects can therefore read the original document path but they cannot modify it.

The third step initializes the $Signatures$ data structure which is used for checking the integrity of the path followed by the document, of the original content of the document, and of the operations executed by the subjects. This data structure is initialized by the $CIG$ as follows: component $Signed_{path}$ is set to $S_{do}(Path_d)$, where $Path_d$ is the path built in the previous step; component $Signed_{doc}$ is set to $S_{do}(d')$; component $Signed_{deltas}$ is set to $S_{do}(Deltas_d)$, where $Deltas_d$ is the structure built in the first step.

Once the package is assembled by including the above described control information and the encrypted document, it is ready to be deployed to the first subject in the $Path_d$ list. Before sending the document, the $EM$ sends to the subjects the keys they need to decrypt the document regions in accordance to the stated access control policies. To compose the correct set of keys to be delivered to each subject in the cooperative group, the $EM$ makes use of the content of the $Key-info$ table and of the $PB$. In particular, each subject receives the key associated with a region, if it satisfies at least one of the policies that apply to that region.

In order to take part in the distributed document update process each subject must have:

1. The keys for each portion of the document it can access according to the stated access control policies, which are used to reconstruct the last version of the document, and to encrypt the new data it adds to the original document.

2. The set of its valid authoring certificates which prove its rights of executing update operations over the document.

3. Its personal private key in order to sign the data that describes the operations it executes and also those executed by the previous subjects, that is, the set of deltas.

4. The public keys of all the other subjects in order to validate the signatures computed over the deltas.

In the following, we present the protocols used to support the process of distributed and cooperative document updates. More precisely, Sections 5.4.1 and 5.4.2 illustrate respectively how
Chapter 5: Cooperative updates in colluding byzantine distributed systems

Section 5.4.3 describes how to execute and keep track of the modification operations to the document, how to register extension operations to the path, that must be followed by the document, and which are the subjects that must be notified to avoid illegal operations due to the collusion of two or more subjects.

### 5.4.1 Path and deltas integrity check

Upon receiving a document package a subject $s$ must first check that it is trustworthy. Since the content of the package contains both the original encrypted document $d'$, signed by the $EM$ with the private key of $do$, and all the updates performed by the previous subjects, the subject will have to:

1. Check the validity of $d'$.

2. Check the validity of successive deltas stored in the $Deltas_d$ data structure.

Check on $d'$ is quite straightforward. The document $d'$ is valid if it is correctly signed by the $EM$. If not, then $s$ can send a recovery request to the $do$ to obtain a correct version of the document.

The deltas validity check can be organized according to two phases. The first phase is described in this section, the second phase, that is, the check of the validity of the updates operated over the document, is described in the next one.

In the first phase $s$ must check the correctness of the path followed by the package wrt both the original path and the extensions operated by the previous subjects that received the package, and the integrity of the deltas stored in the $Deltas_d$ data structure. To do that the protocol makes use of the content of the $Signatures$ data structure.

The first step is to create a local copy of $Path_d$, that will be updated by locally executing the $OP_{EXTENSION}$ operations found in the $Deltas_d$ data structure.

Let $|Deltas_d|$ and $|Signed_deltas|$ be respectively the cardinality of the list of deltas in $Deltas_d$ data structure and the cardinality of the list of signatures in the component $Signed_deltas$ of $Signatures$ data structure, that is, the number of deltas and signatures present in the respective lists. Subject $s$ must check that $|Deltas_d| = |Signed_deltas|$, that is, all the deltas recorded in the package are certified by a specific signature. If this is not the case an error occurs and $s$ must send a
recovery request to the do to manage this exception. Let $s_j$ be the subject in the $j^{th}$ position of the local copy of $Path_d$ (for $j = 1, \ldots, |Deltas_d|$). Subject $s$ must:

1. Compute the hash value over the list of deltas (from $\delta_{do}$ to $\delta_{s_j}$) and check that it is equal to the value stored in the $j^{th}$ position of the $Signed\_deltas$ list, value that must be correctly signed by $s_j$.

2. Check that the $\delta_{s_j}$ content is correct according to Definition 8.

3. Insert in the local copy of $Path_d$ the list of subjects listed in the $OP\_EXTENSION$ operation, if there exists such operation stored in $\delta_{s_j}$, between $s_j$ and the successor subject in the path before this update.

4. Find the tuple $(s_{x}, j)$, if any, stored in its own effective path checklist (see Definition 10). If that tuple exists and $s_x \neq s_j$, then raises a document recovery request. Document recovery can be executed by asking subjects $s_j$ and $s_x$ for their $Deltas_d$ and $Signatures$ data structures. Based on those information, it is quite easy to find out who added different path extension operations for different subjects. The subject who has originated different path extensions, is for sure one of those who colluded to invalidate the integrity of the data. To manage this exception, subject $s$ sends do a recovery request.

5. Check the position of $\delta_{s_j}$ within the $Deltas_d$ data structure. If $\delta_{s_j}$ is the last one stored in that structure and there is at least another tuple $(s_x, x)$ in the effective path checklist associated with the current document, with $x > j$, then an error occurs. An error occurs also if $\delta_{s_j}$ is the last stored in that structure and the subject after $s_j$ in the local copy of $Path_d$ is different from $s$. Also in these cases $s$ sends do a recovery request to manage this exception.

Note that for $j = 1$, $s$ must check that:

- $s_1 = do$
- $\delta_{do} = <> \{(\text{NO\_OP, null})\}$
- $Signed\_deltas = S_{do}(<> \delta_{do})$.

Whenever a subject $s$ in the cooperative group receives a notification $(s_x, x)$ for an XML document $d$, by another subject, $s$ must compare it with the entries stored in its own effective path
checklist regarding $d$. If it finds an entry $(s_y, x)$, with $s_x \neq s_y$ and the same value for the component position, $x$, an error occurs because two different subjects declare to be in the same position in the path. To manage this exception, $s$ sends the do a recovery request.

### 5.4.2 Document integrity check

Besides checking the integrity of the path followed by the document and the integrity of the deltas added by the subjects, a subject $s$ has also to check the validity of the deltas as well. For checking the validity of the operations in a list of deltas, a subject must control each delta in the list. If the operation is of type $OP_{UPDATE}$, then the subject must extract the arguments of the operation that will be in the form $<RUR_1, RUR_2, \ldots, RUR_n>$ and perform a check for every region update record in the list. So, for each $RUR = (\text{region}, UC, AEUL)$, $s$ must check that:

1. The Update Certificate $UC$ is valid for the region specified in the component $\text{region}$.
2. The $\text{subj}_id$ present in $UC$ is equal to the $\text{id}$ of the subject who added the deltas in $\text{Deltas}_d$.
3. The atomic elements specified in the component $AEUL$ are present in the $prot_id$ field of $UC$.

By performing the above steps, $s$ can be sure that:

- Each operation described in the $\text{Deltas}_d$ data structure was performed by the subject that signed and certified the modification.
- Each operation described in the $\text{Deltas}_d$ data structure was executed according to the access control policies in $\mathcal{PB}$.
- No operations were executed by unauthorized subjects.

Once subject $s$ has checked that either the original document and the subsequent modifications are trustworthy, it can obtain the last version of the document by reading all the deltas present in the $\text{Deltas}_d$ data structure and locally performing the corresponding modifications on the original document. Note that, each element in $d$ and each update in the $\text{Deltas}_d$ data structure are encrypted according to access control policies defined by the server. In this way each subject can only read the elements, and the modifications to the elements, that it is authorized to read. This assures that the data in $\text{Deltas}_d$ cannot be exploited by breaking the protocol and only the subjects who possess the right keys can access the data, either original or modified, of a given document region.
5.4.3 Protocol for document and path modification

When a subject $s$ executes some modifications to the document content or to the path, it has to register these operations in the $Deltas_s$ data structure. In particular, the protocol generates a new entry, $\delta_s$, in the $Deltas_s$ data structure in which the description of all the executed modifications must be inserted, that is, an atomic delta for each type of executed operation as described in Definition 8.

If a subject $s$ modifies the document content, the modification must be described as deltas with respect to the last version of the document that the subject has locally rebuilt, and recorded in $\delta_s$. For this purpose, the protocol generates a new atomic delta, inserted in $\delta_s$, with value $OP\_UPDATE$ in its component $op\_type$, and inserts in the component $op\_arg$ a set of RURs, one for each region involved in the updates. The protocol takes also care of inserting in each generated RUR the correct set of modified elements, all belonging to the same region specified in the component $region$, and the corresponding update certificates.

By contrast, if $s$ wishes to add a set of subjects to the current path, the protocol has only to generate a new atomic delta with value $OP\_EXTENSION$ in its component $op\_type$, and inserts in the component $op\_arg$ the list of subjects that will have to receive the document. Also in this case the generated atomic delta must be inserted in $\delta_s$.

Example 7 With respect to the content of the data structures presented in Figure 5.4, we show in Figure 5.5 the graph representation of the $OP\_EXTENSION$ operation executed by subject 154. On the left of the figure there is the path before the extension operation, whereas on the right there is the updated path. The subject to which 154 has to send the package is 138, that is, the first subject of the path extension.

Note that even if a subject does not execute any modification over the document or over the path, it must generate and insert a new atomic delta in the $\delta_s$. This atomic delta contains in the component $op\_type$ the value $NO\_OP$ and in the component $op\_arg$ the value $null$. This is done in order to certify that the path has been followed since the current subject.

Subject $s$ has also to compute the signature over the current content of the $Deltas_s$ data structure, that will be appended to the $Signed\_deltas$ component of the $Signatures$ data structure.

When the update phase is finished, $s$ sends the document package to the successor subject in the path or to the first subject of the list of subjects specified for an $OP\_EXTENSION$ operation that it has previously inserted in the $\delta_s$. 
The protocol is robust against the collusion of at most $k$ subjects, where $k$ is the security level established by $do$ at the beginning of the update process, and represents the maximum number of subjects that can collude without affecting the protocol.

To ensure this property, if subject $s$ belongs to an extension of the original path, at any level of nesting, it has to notify its position in the path to a set of subjects belonging to the cooperative group. The notification signed by $s$ consists of the identifier of $s$ and of an integer representing the position ($pos$) occupied by $s$ in the path (i.e., $S_s(s, pos)$). The set of subjects to which this notification must be sent is composed by the first $n$ distinct subjects of each extension in the path, that follows $s$, where $n - 1$ is the security level associated with the protocol. This means that if $s$ belongs to the $k^{th}$ extension, it has to send the notification to: 1) the first $n$ distinct subjects that follow $s$ belonging to his/her extension; 2) the first $n$ distinct subjects that follow $s$ in the path belonging to the extension at the upper level and so on until the first $n$ subjects that follow $s$ in the original path.

If $s$ has executed a $OP_{EXTENSION}$ operation it has to send the notification also to the first $n$ subjects belonging to the extension corresponding to that operation. It is also needed sending the notification to the $do$ if in the original path ($Path_d$) there are less than $n$ distinct subjects that follow $s$. If one of the considered extensions contains less than $n$ distinct subjects following $s$, the protocol sends the notification only to the subjects that are present.

**Example 8** Figure 5.6 shows a possible scenario of document flow (on top of the Figure) and reports, for each involved subject, the content of its local effective path checklist data structure (in
Chapter 5: Cooperative updates in colluding byzantine distributed systems

83

Figure 5.6: Effective path checklist data structures, when the security level is set to 2

<table>
<thead>
<tr>
<th>Subject</th>
<th>Effective Path Checklist</th>
</tr>
</thead>
<tbody>
<tr>
<td>do</td>
<td>{ (122, 4), (112, 5), (102, 6), (134, 7), (168, 8) }</td>
</tr>
<tr>
<td>secretary s_id = &quot;102&quot;</td>
<td>{ (102, 2), (112, 5), (102, 6) }</td>
</tr>
<tr>
<td>accountant s_id = &quot;104&quot;</td>
<td>{ (104, 3) }</td>
</tr>
<tr>
<td>manager_company_director s_id = &quot;158&quot;</td>
<td>{ (122, 4), (112, 5), (102, 6), (134, 7), (168, 8), (158, 9) }</td>
</tr>
<tr>
<td>accountant s_id = &quot;122&quot;</td>
<td>{ (122, 4) }</td>
</tr>
<tr>
<td>manager s_id = &quot;112&quot;</td>
<td>{ (122, 4), (112, 5), (122, 4) }</td>
</tr>
<tr>
<td>accountant s_id = &quot;134&quot;</td>
<td>{ (122, 4), (112, 5), (102, 6), (134, 7) }</td>
</tr>
<tr>
<td>manager s_id = &quot;168&quot;</td>
<td>{ (122, 4), (112, 5), (102, 6), (134, 7), (168, 8) }</td>
</tr>
</tbody>
</table>

In case the security level is set to 2, i.e., the collusion of two subjects must be avoided. CIG inserts in the package the path: \(<\text{do}, 102, 104, 158>\), then it sends the package to the first subject (102). During the document flow, subject 104 executes an \textit{OP\_EXTENSION} operation inserting in the original path subjects: \(<122, 112, 134, 168>\). Subject 112 executes the same operation inserting the extension: \(<102>\). Below each subject belonging to an extension, independently by its level of nesting, Figure 5.6 reports the set of subjects to which it has to send its notification, following the protocol explained above.

Each subject must also locally store the modifications to the document that it has made. If a request of document recovery is originated by a subject, the \textit{do} will ask all the deltas that each subject has added during the update process, build an updated version of the document locally, and start again the protocol from the subject that has originated the document recovery request.
Chapter 5: Cooperative updates in colluding byzantine distributed systems

5.5 A running example

In this section we report a possible scenario of a malicious operation executed by two colluded subjects, and we show how it can be detected by our protocol.

Example 9 In Figure 5.7 we show the same scenario presented in the Example 8, where the CIG sends a document associated with the following path: $Path_d = \langle do, 102, 104, 158 \rangle$. The security level in this case is set to 2, that is, we wish to be protected against collusion of at most two malicious subjects. During the update process subject 104 adds a path extension with subjects $\langle 122, 112, 134, 168 \rangle$ (depicted as grey rectangles, differently from the subject belonging to $Path_d$, depicted with white rectangles) and then also subject 112 adds an extension with only subject 102. In the graph representation of the path, under each subject there is the set of subjects to which a notification must be sent, according to the protocol presented in Section 5.4.3. The table shown at the bottom of the Figure reports the effective path checklist for all the subjects taking part in the document flow.
Suppose that subjects 104 and 102, highlighted in the picture with an hatching background, collude in order to delete the deltas added by subjects 122 and 112, to exclude from the update process subjects 134 and 168 and add subjects 138 and 154 to the path that the document must follow. To do that 104 generates a new extension to the path containing subjects: 102, 138 and 154, inserts it in its delta\(_{104}\) and then it signs the new content of \(\delta_{\text{d}}\). The package is then updated by 102, that inserts its delta\(_{102}\) in the \(\delta_{\text{d}}\) data structure, signs that structure and then sends the package to 138. Subject 102 does not send its notification to subjects 158 and 168 trying to hide this malicious operation. Subjects 158 and 168 detects this operation when 138 sends them its notification with value 5 for the component position, because they have received another notification with that value \((112, 5)\). Even if 138 and 154 do not send their notifications to subjects 158 and 168, when 158 receives the document it finds in the forth position of the path 102, whereas it has received the notification \((122, 4)\), that is, the declaration of subject 122 to have occupied the forth position in the path for that document, detecting the executed malicious operation.

5.6 Correctness proof

In this section we show some correctness results for our protocol. An important property that the protocol must ensure is that, no matter how many malicious subjects collude, it must not be possible to modify the document in such a way that it is not possible to identify the originator of the modifications. Additionally, it must be ensured that operations deleting an update performed by a non-malicious subject can be detected.

Since it is impossible to modify the content of the original document, \(d\), because it is signed by the \(EM\), we will focus on the possible tampering operations of the other data structures: \(\delta_{\text{d}}\) and \(\text{Signatures}\).\(^2\)

In the following, a list of tampering cases is provided. For each of these cases we show that the protocol gives a subject enough information to detect the tampering. In particular, we focus on the following cases:

1. A subject performs an update for which it does not possess the proper authorization.

2. Deltas tampering:
   
   (a) Erasing of a \(\delta_{\text{d}}\) data structure.

\(^2\)The \(Path_{\text{d}}\) data structure is unbreakable since it is signed by the \(CIG\).
(b) Modification of a $delta_s$ data structure.


1) A subject performs an update for which it does not possess the proper authorization.

In this case, the malicious subject tries to add an $OP_{UPDATE}$ operation without a valid $UC$ or with a $UC$ stolen to another subject. In the first case, the other subjects, by inspecting the $UC$, can detect that the update privilege is not appropriate or that the prot$_id$ component of the certificate does not match the document region and/or the atomic element on which the operation is applied.

In the second case, the subjects receiving the package will see that, the subject specified in the $UC$ is different from the one which has signed the information containing this $UC$.

2.a) Erasing of a $delta_s$ data structure.

This is the case of a malicious subject trying to completely erase the $delta_s$ data added in the $Deltas_d$ data structure by a previous subject $s$. In this case, the deletion of the $delta_s$ and the corresponding signature computed by $s$ over this information and all the other deltas already present in the $Deltas_d$ data structure will be detected by the next subject because all the signatures generated and inserted in the package by the subjects that have received the package after $s$ and before the malicious subject were computed also over $delta_s$. Also the collusion between all the subjects in the path between $s$ and the malicious one is not enough to hide the elimination of $delta_s$ to the next subjects, because in the path the presence of $s$ is still required. In particular, if $s$ is a subject belonging to the original path, it is not possible to delete its presence from the path. It is possible to do that only if $s$ belongs to an extension of the path and among the subjects that collude there is the subject that has executed that extension to the path or an extension that contains that one, but the process of notification ensures that this deletion is detected by at least one non-malicious successor subject. In general, we consider the case in which there is a collusion among at most $n-1$ subjects that have already received the package. For the reason explained before one of them must be a subject that precedes $s$ in the path. All the other $n-2$ malicious subjects must follow $s$ in the path and can be positioned in any point of the path. Since $s$ has notified $n$ distinct subjects in every extension at any level of nesting we can conclude that, among the subjects that follow it in the path, there exists at least one non-malicious subject that has received the notification and that can detect the elimination of subject $s$ from the path and of all the information associated with it.

2.b) Modification of a $delta_s$ data structure.

In this case, a malicious subject tries to modify the $delta_s$ list added in the $Deltas_d$ data structure
by a previous subject. This form of tampering is quite easy to detect because the malicious subject should have all the private keys of the subjects, starting from the subject who added the \( \text{delta}_k \) to the subject before the malicious one, in order to modify their signatures accordingly to the new content of the \( \text{Deltas}_d \) data structure and so to hide the tampering action. In particular the malicious subject will not be able to obtain the private key of the damaged subject, and so any modification will be simply detectable.

3) Document path tampering.
A malicious subject can try to modify the intended document path in two ways: 1) changing the \( \text{Path}_d \) data structure (unfeasible); or 2) modifying the extension operations added by previous subjects. This second case is a particular case of the 2.b.
Chapter 6

Cooperative updates in colluding byzantine and failure prone distributed systems

Contents

6.1 Introduction ........................................ 91
6.2 Overview ............................................... 92
  6.2.1 Document Originator ............................. 93
  6.2.2 Cooperative Group ................................ 93
  6.2.3 Delegates Group .................................. 94
6.3 Specification Languages ............................... 94
  6.3.1 Flow Policy Specification Language .......... 94
  6.3.2 Flow Modification Specification Language ...... 96
6.4 Control Information .................................... 100
  6.4.1 Preliminary Concepts ............................. 101
  6.4.2 Document Control Information .................. 106
  6.4.3 Modification declarations ....................... 109
6.5 Distributed and cooperative update process .......... 111
  6.5.1 Assumptions and Properties ..................... 112
  6.5.2 Protocol Parameters Setting ..................... 115
Chapter 6: Cooperative updates in colluding byzantine and failure prone distributed systems

6.5.3 Update process phases

6.5.3.1 Initialization phase

6.5.3.2 Cooperative update phase

6.5.3.3 Final phase

6.5.4 Formal results

6.5.5 Recovery

6.5.5.1 Preliminary definitions

6.5.5.2 Recovery algorithm

6.5.5.3 Recovery responses merge algorithm

Figures

6.1 (a) An example of XML flow policy and (b) its graph representation

6.2 An example of Rule Base

6.3 Message exchange

6.4 Sub-phases messages exchange

6.5 Document Originator Protocol

6.6 Subject Protocol

6.7 Delegate Protocol

6.8 Down Delegate Protocol

6.9 The recovery algorithm (part 1)

6.10 The recovery algorithm (part 2)

6.11 An Algorithm for the merging of recovery responses

Tables

6.1 Structure of a flow policy

6.2 Components of the structure of a flow policy

6.3 Possible configurations of a flow modification rule and their semantics

6.4 Table containing flow policy regions for the flow policy in Figure 6.1

6.5 Control data structures for deletable and non-deletable document atomic elements

6.6 Components of the control data structures for deletable and non-deletable document atomic elements

6.7 Control data structures for non-modifiable document regions
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.8 Components of the control data structures for non-modifiable document regions</td>
<td>107</td>
</tr>
<tr>
<td>6.9 Modifiable Region Classification</td>
<td>108</td>
</tr>
<tr>
<td>6.10 Control data structures for modifiable document regions</td>
<td>108</td>
</tr>
<tr>
<td>6.11 Components of the control data structures for $FDU R$</td>
<td>109</td>
</tr>
<tr>
<td>6.12 Modification declaration structure within a receiver specification</td>
<td>110</td>
</tr>
<tr>
<td>6.13 Components of the Modification Declaration Structure</td>
<td>111</td>
</tr>
<tr>
<td>6.14 Structure of the Invalid Modification Document Declaration information</td>
<td>113</td>
</tr>
<tr>
<td>6.15 Data structures used by the protocols</td>
<td>114</td>
</tr>
<tr>
<td>6.16 Messages (part 1)</td>
<td>118</td>
</tr>
<tr>
<td>6.17 Messages (part 2)</td>
<td>119</td>
</tr>
<tr>
<td>6.18 Functions (part 1)</td>
<td>127</td>
</tr>
<tr>
<td>6.19 Functions (part 2)</td>
<td>128</td>
</tr>
</tbody>
</table>
Chapter 6: Cooperative updates in colluding byzantine and failure prone distributed systems

6.1 Introduction

In this chapter we present an approach for the distributed and cooperative updates of XML documents, supported by our framework, particularly suited for colluding byzantine and failure prone distributed systems. We recall that with the term byzantine we mean a party involved in the process that do not obey the corresponding protocol. The proposed approach is based on a specific infrastructure, that enrich that already presented in Chapter 3. We believe that two are the main additional requirements of such an infrastructure.

The first is the development of a suite of high level languages for flow policy, and flow modification specification. The language for the flow policy specification has to enable a subject, denoted in the following as the flow policy originator (fpo), possibly different by do, to partially or totally specify the path that the document must follow in terms of the characteristics that have to be possessed by the receivers. Further such a language has to enable a flow policy originator to directly specify into a flow policy whether a receiver can or cannot modify the path that the document must follow. A flow policy is the equivalent of the concept generically introduced in Chapter 3 as path conditions. By contrast language for flow modification is used to specify which subjects can modify which portions of a flow policy and in which mode. More precisely, the flow modification specification language is used by the flow policy originator of a flow policy (fp) to generate flow modification rules that apply to fp.

The second requirement is the development of enforcement mechanisms able to prevent subjects from executing undetectable illegal operations over the flow policy and/or document content. In particular, a subject knows which privileges it can exercise and over which portions of a flow policy or of an XML document, because it receives from the corresponding originator some certificates, generated respectively according to the specified flow modification rules and access control policies, attesting the rights it possesses on the flow policy and/or XML document. The possibility of modifying both a flow policy and an XML document requires the generation of some specialized control information to properly exercise the privileges specified by our two specification languages (flow modification and access control policy specification languages) and to make it possible for a subject to check the integrity of a flow policy and of the corresponding received XML document.

Control information together with its corresponding flow policy form the so-called flow policy attachment. Finally, we recall that an XML document together with its corresponding control information forms a document package. Note that the generation of a flow policy is independent from its association with a particular XML document and therefore a flow policy can be reused and
associated with different XML documents.

Parties involved in the approach presented in this chapter for the distributed and cooperative updates of XML documents are: a Cooperative Group, denoted also here as \( CG \), a Delegates Group, denoted as \( DG \), and the \( do \) denoted also here as \( do \). The additional party involved in this approach, the Delegates Group, is the set of subjects, called delegates, chosen by the \( do \) at the beginning of the process to execute, at each step of the update process, the flow policy integrity check and, whenever required by a subject, the document content recovery.

Our approach achieves the following goals: 1) to allow a subject to view all and only the document portions to which it has access; 2) to allow an authorized subject to modify both the flow policy and the document content according to the corresponding flow modification rules and access control policies respectively; 3) to allow a subject to detect illegal operations executed over the document content by a single byzantine subject or by a set of colluding byzantine subjects; and 4) to allow delegates to detect illegal operations executed over the flow policy by a single subject and to recover corrupted documents. These goals are achieved through the execution of a suite of protocols, each of them particularly suited for the specific party to which it corresponds to, that are invoked during the cooperative and distributed update process in well specified moments called, in the following, phases of the process.

The remainder of this chapter is organized as follows. Section 6.2 gives an overview of the approach presented in this chapter. Section 6.3 introduces the specification languages used to specify both flow policies and flow modification rules. In Section 6.4 we present control information required by our protocols to reach the above-mentioned goals. Section 6.5 presents all the phases and corresponding sub-phases, if any, that occurs during the distributed and cooperative update process, according to the approach described in this chapter, and directly refers, during the description of such (sub)phases, to the protocols executed by the involved parties. Finally, Section ?? concludes this chapter presenting the Recovery sub-phase, that is one of the most innovative aspects of the proposed approach.

### 6.2 Overview

In this section we briefly describe the interaction between the involved parties and define some basic concepts on which our approach relies. The document generated by \( do \) is sent to a first chosen subject in \( CG \), then the document flows among the other subjects, according to the flow policy attachment content, and finally it is returned to \( do \). Whenever an error occurs to the
document content the current subject \( s_c \), that is the last subject that has received the document to be updated, contacts all delegates in \( DG \) to start a Recovery sub-phase. At the end of this sub-phase \( s_c \) obtains the last correct version of the document and thus can update the document according to its modification rights. During the Recovery sub-phase delegates interact with subjects in \( CG \) to obtain the last correct version of the document and build the recovered document to be sent to \( s_c \).

We call step all the operations/interactions executed by \( s_c \) and delegates from the reception of the document and flow policy attachment by \( s_c \), to the delivery of the updated document to the next receiver \( s_{next} \). The complete cooperative and distributed update process thus consists of a set of steps.

Following Sections 6.2.1, 6.2.2, and 6.2.3 give more details about each of the above-mentioned parties.

6.2.1 Document Originator

In this approach \( do \) is the subject who generates the XML document package \( (Doc_{do}) \) to be updated and the associated flow policy attachment \( (Fpa_{do}) \). This subject also specifies the set of access control policies that applies to \( Doc_{do} \) and the set of flow modification rules that applies to \( Fpa_{do} \), generates and distributes the corresponding document/flow policy modification certificates and document decryption keys to the proper subjects and starts the update process. At the beginning it sets two parameters, \( (b) \) and \( (d) \), such that \( b \) is the maximum number of byzantine delegates that do not affect the protocol, and \( d \) is the maximum number of down delegates that do not delay the protocol. These two parameters are used to determine the exact cardinality of \( DG \).\(^1\) Moreover the \( do \) chooses the set of delegates that will manage the update process and the set of subjects that will be able to take part in the cooperative update process (members of \( CG \)). Furthermore, before sending \( Doc_{do} \) and \( Fpa_{do} \) to the first chosen receiver in \( CG \) and thus starting the update process, the \( do \) distributes them also to all delegates in \( DG \) that will use them during the \( Fpa\)-Checking and Recovery sub-phases.\(^2\)

6.2.2 Cooperative Group

Subjects belonging to \( CG \) are chosen at the beginning of the process, and receive in advance all the information (certificates and decryption keys) required to exercise the privileges they

---

\(^1\)More details can be found in Section 6.5.1.
\(^2\)More details can be found in Section 6.5.3.
Chapter 6: Cooperative updates in colluding byzantine and failure prone distributed systems

posses over the document/flow policy attachment and to decrypt document portion they can access. Moreover, they are the only ones that can be chosen to be receivers of the XML document. CG can contain an unlimited number of byzantine subjects.

6.2.3 Delegates Group

DG is a set of subjects chosen by the do at the beginning of the update process to check the flow policy integrity, at each step of the process, and, whenever required by a subject, to execute document content recovery. The set of delegates is partitioned into three subsets: the set of byzantine delegates (B, with |B| ≥ 0), the set of operative delegates (OP, with |OP| ≥ 0), and the set of down delegates (D, with |D| ≥ 0). More precisely, operative delegates obey the protocol and are reachable by the subjects, whereas down delegates are unreachable or reachable but not able to correctly accomplish their job. Only after a Down-Delegate-agreement sub-phase a down delegate becomes operative once again, whereas an operative delegate becomes down whenever a failure occurs.

6.3 Specification Languages

In this section we present two of the three main specification languages that we need to support a cooperative and distributed update of XML documents. In particular Section 6.3.1 deals with the flow policy specification language, Section 6.3.2 introduces our language for expressing flow modification rules, whereas the language used to express access control policies has been already presented in Section 3.3.

6.3.1 Flow Policy Specification Language

A flow policy denotes the sequence of subjects that must receive the package to which it is associated. This sequence can be fully specified in advance, at the beginning of the update process, or partially specified when the process starts and then modified and extended by authorized subjects. A flow policy does not necessarily contain the fully specified list of receivers, rather it can contain some receiver specifications, that is properties that have to be possessed by the receivers. Such properties are specified by means of the so called credentials. Figure 3.3 in Chapter 3 shows an example of credentials.

Each receiver specification contains one or more alternative receiver profiles. A receiver
Chapter 6: Cooperative updates in colluding byzantine and failure prone distributed systems

Notation | Structure | Semantics
--- | --- | ---
Fpa | (...)\ FlowPolicy, ...) | Flow Policy Attachment. It consists of a (sub)flow policy and the corresponding control information. It is equal to null when a receiver does not insert a sub flow policy in its ReceiverSpec

FlowPolicy | <ReceiverSpec1, ... ReceiverSpecN> | Ordered list of receiver specifications specified by the Originator of the Flow Policy Attachment

ReceiverSpeci | (rs\_id, Profiles, ... Fpa) | A receiver specification

Profiles | set of ReceiverProfile | Set of alternative receiver profiles. A subject must satisfy at least one of them to be a valid receiver

ReceiverProfile | (rp\_id, CredSpec, ExtSpec) | Single receiver profile

CredSpec | (cs\_id, CredExpr) | Credential specification component of an alternative profile

ExtSpec | (es\_id, ExtExpr) | Extension component of an alternative profile

Table 6.1: Structure of a flow policy

satisfies a receiver specification if it satisfies at least one of the receiver profiles contained in that specification. Receiver profiles consist of a credential expression, that is a condition specified against credentials by means of XPath. Our flow policy specification language enables also an originator to grant a receiver the permission of extending a flow policy inserting a sub flow policy. This is obtained by adding an additional component to the receiver specification called ExtExpr, which assumes the value subpath, if an insertion is enabled, or nosubpath, if an insertion is denied. Table 6.1 presents the structure of a flow policy, whereas Table 6.2 gives the semantics of each component of a flow policy.

A sub flow policy is a flow policy inserted within the receiver specification associated with \( s_c \). Let \( s_{sc} \) be the subsequent receiver of \( s_c \) in the flow policy before the insertion of a sub flow policy, then receivers specified in the receiver specifications contained in such a sub flow policy must receive the document after \( s_c \) and before \( s_{sc} \).

Example 10 The following is an example of flow policy:

\[ <(rs\_id1 \{ (rp\_id2, (cs\_id3, "//manager[@department="R&D"]"), (es\_id4, subpath)), (rp\_id5, (cs\_id6, "//secretary[@department="R&D"]"), (es\_id7, nosubpath))\}, \]

...
Component | Semantics
--- | ---
rs_id, rp_id, cs_id, es_id | receiver specification, receiver profile, credential specification and extension specification identifier
CredExpr | conditions that must be satisfied by the credentials of a receiver
ExtExpr | value in \{subpath|nosubpath\}, stating whether a receiver is enabled (subpath) or not (nosubpath) to insert a sub flow policy in a flow policy

Table 6.2: Components of the structure of a flow policy

\{(rs_id8, \{(rp_id9, (cs_id10, "//notary[@law_firm='FLYNN & FLYNN']")), (es_id11, subpath))\}),
\{(rs_id12, \{(rp_id13 (cs_id14, "//company.management.director"), (es_id15, subpath))\}))\} |

which specifies that the first receiver must be a manager or a secretary of the "R&D" Department; the second receiver must be a notary of the "FLYNN & FLYNN" law firm; whereas the third receiver must be a company management director. Moreover the flow policy specifies that whereas managers, notaries and company management directors are entitled to insert a new sub flow policy into this flow policy, secretaries are not enabled to do that.

Figures 6.1 (a) and (b) show respectively the flow policy and its graph representation, corresponding to the flow policy presented in Example 10.

6.3.2 Flow Modification Specification Language

Modification of flow policies are governed by flow modification rules, which state who can modify a flow policy. In this section, we introduce the language we propose to specify flow modification rules. In particular, we focus on the main components of a rule, that is how subjects and protection objects are qualified in the rules, which privileges can be specified in a rule, and finally which types of propagation options we provide.

**Subjects.** In order to make more flexible the specification of the set of subjects to which a rule applies, the subject specification, contains a credential expression.

**Protection objects.** A protection object is a portion of a flow policy to which a flow modification rule applies. We provide a wide range of possibilities in the specification of a protection object. In particular the language allows one to specify rules that apply to a set of flow policies, a single flow policy, and selected portion(s) of a flow policy.

Moreover, it is possible to apply a flow modification rule to one of the previous objects by taking into account its content in addition to its structure. This is obtained by using an XPath-
Figure 6.1: (a) An example of XML flow policy and (b) its graph representation
compliant language [39], to specify the protection objects to which a rule applies. A flow modification rule can only apply to selected elements of a flow policy, namely ReceiverSpec, ReceiverProfile, CredExpr, and ExtExpr elements, because the other elements contain control information.

**Privileges.** Flow modification rules can refer to two different privileges, namely update and delete. The first one allows a subject $s$ to modify only the data content of a CredExpr and/or ExtExpr element of a flow policy. This means that $s$ can modify the alternative properties that a receiver subject has to possess to be a valid receiver and/or it can enable or disable a receiver to insert a sub flow policy. The second privilege gives a subject the possibility of deleting one or more ReceiverSpec and/or ReceiverProfile elements in a flow policy. This operation allows a subject to modify the path that the package has to follow, by reducing the number of subjects that will have to receive it or deleting some of the alternative profiles that compose a receiver specification. We do not provide an explicit add privilege, because this information is directly inserted in the flow policy by the originator in the ExtExpr elements. Furthermore, the originator can also give to other subjects the right of enabling or disabling the insertion of a sub flow policy, generating proper modification control rules referring to the update privilege, as explained above.

**Propagation options.** A propagation option specifies whether and how a flow modification rule specified on a given protection object propagates to other protection objects in the same flow policy. We support two types of propagation options: NO PROP and PROP. The NO PROP option means that a rule applies only to the protection objects specified in the flow modification rule itself, whereas the PROP option propagates the effect of a flow modification rule to all the protection objects belonging to the sub-trees rooted at the protection objects specified in the flow modification rule itself. The full semantics of these propagation options is given in Table 6.3, that also states the constraints\(^3\) on protection objects, privileges, and propagation options that a flow modification rule must satisfy.

We are now ready to formally define a flow modification rule in terms of the components presented so far.

**Definition 11 (Flow Modification Rule).** A flow modification rule $\text{fmr}$ is a tuple $(r\text{id}, \text{credExpr}, \text{target}, \text{path}, \text{priv}, \text{propOpt})$, where: a) $r\text{id}$ is the rule identifier; b) $\text{credExpr}$ is a credential expression; c) $\text{target}$ and $\text{path}$ specify the protection objects to which the rule applies; more specifically target specifies one or more flow policies, whereas path is an XPath-compliant expression denoting specific portions within the target; d) $\text{priv} \in \{\text{update, delete}\}$ is one of the supported privileges;

\(^3\)The symbol “$|$” is used to denote an alternative choice.
### Chapter 6: Cooperative updates in colluding byzantine and failure prone distributed systems

<table>
<thead>
<tr>
<th>Priv.</th>
<th>Protection object</th>
<th>P. opt.</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>update</td>
<td>set of flow policies</td>
<td>PROP</td>
<td>The privilege is applied to all the data contents contained in the elements $CredExpr$ and $ExtExpr$ belonging to the specified flow policies, allowing a subject to modify them</td>
</tr>
<tr>
<td></td>
<td>a single flow policy</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>set of $ReceiverSpec$ elements</td>
<td>PROP</td>
<td>The privilege is applied to all the data contents contained in the $CredExpr$ and $ExtExpr$ elements belonging to the subtrees rooted in the protection objects specified, allowing a subject to modify them</td>
</tr>
<tr>
<td></td>
<td>set of $ReceiverProfile$ elements</td>
<td></td>
<td></td>
</tr>
<tr>
<td>delete</td>
<td>set of flow policies</td>
<td>PROP</td>
<td>The privilege is applied to all the $ReceiverSpec$ and $ReceiverProfile$ elements contained in the specified flow policies, allowing a subject to delete one or more of these elements</td>
</tr>
<tr>
<td></td>
<td>single flow policy</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>set of $ReceiverSpec$ elements</td>
<td>NOPROP</td>
<td>The privilege is applied only to all the $ReceiverSpec$ elements contained in the specified flow policies, allowing a subject to delete one or more of these elements</td>
</tr>
<tr>
<td></td>
<td>set of $ExtExpr$ elements</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>set of $CredExpr$ elements</td>
<td>NOPROP</td>
<td>The privilege is applied only to all the specified protection objects and to all the $ReceiverProfile$ elements belonging to the subtrees rooted at these objects, allowing a subject to delete one or more of these elements</td>
</tr>
<tr>
<td></td>
<td>set of $ExtExpr$ elements</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NOPROP</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.3: Possible configurations of a flow modification rule and their semantics
Chapter 6: Cooperative updates in colluding byzantine and failure prone distributed systems

100

\[
\begin{align*}
\text{Figure 6.2: An example of Rule Base}
\end{align*}
\]

e) \(\text{prop\_opt} \in \{\text{NO\_PROP, PROP}\} \) is one of the two possible propagation options.

Similar to credentials, also flow modification rules are encoded using XML. We denote with the term Rule Base (\(RB\)) an XML file encoding a set of flow modification rules.

**Example 11** Figure 6.2 shows a rule base referring to the flow policy attachment reported in Figure 6.1 (FlowPolicy_Department-monthly_report.xml). According to the rules in Figure 6.2 managers working in the "R&D" department are entitled to delete all the ReceiverSpec elements and all its subelements; secretaries working in the "R&D" department are enabled to modify all the data contents belonging to each CredSpec/ExtSpec element; notaries working in the "FLYNN & FLYNN" law firm can delete the ReceiverSpec element with identifier equal to 12 and all its subelements.

### 6.4 Control Information

In this section we introduce the control information required, to subjects to check document content integrity and correctly exercise their modification rights on the document content. We do not present flow policy control information, required to allow delegates to check flow policy attachment integrity and to subjects to correctly exercise their modification rights on the flow policy.
because it is very similar to the document control information and its use and modification follow
the same strategy described for presented document control information. By contrast, we introduce
the concept of modification declarations, that is the history of the modification operations executed
by the subjects on the document and on the flow policies, used both for document and flow policies
during their content integrity check.

Before presenting control information for document and modification declarations, we
have to introduce some preliminary concepts, such as flow policy atomic elements and certificates,
used in the control information definition and by the subjects to correctly exercise their modification
rights.

6.4.1 Preliminary Concepts

Using an approach similar to that followed for the documents, the application of the flow
modification rules, stored in the $RB$, to a flow policy generates a marking of the flow policy itself.
Basic portions of a flow policy to which is associated a label during the marking and that are in-
dependently modifiable (updatable and/or deletable) according to the rules in $RB$ are called flow policy atomic elements. We can formally define a flow policy atomic element as follows.

Definition 12 (Flow Policy Atomic Element). Let $fp$ be a flow policy. Let $e$ be an element belonging to $fp$ and let $id$ be the value of the identifier associated with $e$. The set $FPAE(fp)$ of flow policy atomic elements of $fp$ is defined as follows: 1) for each element $e$ in $fp$, $id$.tags&attrs $\in FPAE(fp)$, where $id$.tags&attrs denotes the tags of $e$ together with its attributes; 2) for each element $e$, such that $e$ contains data content, $id$.dc $\in FPAE(fp)$.

Example 12 Example of flow policy atomic elements in the flow policy attachment in Figure 6.1 are:

a) $1$.tags&attrs = "<ReceiverSpec Id = "1"/> </ReceiverSpec>", tags of the ReceiverSpec element with identifier equal to 1 and corresponding attribute;

b) $5$.tags&attrs = "<ReceiverProfile Id = "5"/> </ReceiverProfile>", tags of the ReceiverProfile element with identifier equal to 5 and corresponding attribute;

c) $3$.dc = "//manager[@department = "R&D"]", data content of the CredSpec element with identifier equal to 3

A marking for a flow policy $fp$ is a set of pairs $(fpae, R)$, where $fpae \in FPAE(fp)$ is a flow policy atomic element of $fp$, and $R$ is a set of flow modification rule identifiers (possibly empty)
that apply to \(fpae\). Also in this case all flow policy atomic elements with the same label compose the so called flow policy region. We assume that each flow policy region is uniquely identified by an identifier. In the following, given a flow policy \(fp\) we denote with \(FPR(fp)\) the set of identifiers of the flow policy regions of \(fp\) implied by the rules in \(RB\). Flow policy regions are generated because, like the corresponding document regions, they are protected by specific control information used to check the integrity and authenticity of the flow policy atomic elements belonging to them.

**Example 13** Table 6.4 shows the set of flow policy regions and correlated information associated with the flow policy in Figure 6.1, according to the policies in Figure 6.2.

The do of an XML document and/or of a flow policy generates, according to its \(PB\) and \(RB\), a set of certificates signed with its private key, containing information concerning the privileges a subject can exercise over some portions of a document and/or flow policy. Certificates generated for XML documents are called document modification certificates, whereas those for flow policies are called flow policy modification certificates.

Both document modification and flow policy modification certificates are used by a subject, that has modified a document/flow policy portion, to prove its right to modify that portion to the subsequent receivers of the package. Therefore, whenever a subject executes a modification it has to add the proper certificate to the control information associated with the modified portion. Before formally introduce the notion of document modification certificate we need to introduce the notion of document protection object, that is the formal specification of the maximum set of document portions on which a subject can independently exercise a possessed modification right according to the stated access control policies. More precisely it consists of a region identifier, if the authorized subject can update/delete each subset of the attributes belonging to that region; or consists of a set of region identifiers, each of them associated with a subset of their atomic elements, if a subject is authorized to delete all the atomic elements belonging to a document sub-tree.
Definition 13 (Document Protection Object). Let \( d \) be an XML document and let \( p \) be a privilege supported by our access control specification language. A document protection object has one of the following forms: 1) \( \text{r.id} \), a document region identifier belonging to \( DR(d) \), if \( p \in \{ \text{update.attr}, \text{delete.attr} \} \), or 2) \( \text{root.id} \) and \( \text{reg} \), where \( \text{root.id} \) is a document atomic element root of a sub-tree of \( d \), whereas \( \text{reg} \) is a set of pairs \( (\text{id}, \text{docae}) \), with \( \text{id} \) document region identifier belonging to \( DR(d) \), and \( \text{docae} \) set of document atomic elements belonging to \( \text{id} \) and to the sub-tree rooted at \( \text{root.id} \), if \( p \) is equal to \( \text{delete.elem} \).

We are now ready to formally introduce the notion of document modification certificate as follows.

Definition 14 (Document Modification Certificate). Let \( do \) be a \( do \), and let \( d \) be an XML document managed by \( do \). Let \( \text{Auth}_P(d) \subseteq PB \) be the set of access control policies, containing an authoring privilege, specified by \( do \) that apply to \( d \) and let \( acp \) be a policy in \( \text{Auth}_P(d) \). Let \( \text{Sbj PK}(acp) \) be the set of public keys of subjects authorized to modify \( d \) according to \( acp \). A document modification certificate \( dmc \), generated according to \( acp \), is a tuple \( (\text{cert.id}, \text{doc.id}, \text{priv}, \text{sbj.pk}, \text{obj}, \text{signature}) \), where: \( \text{cert.id} \) is the certificate identifier that univocally identifies a document modification certificate among those generated by \( do \); \( \text{doc.id} \) is the identifier of \( d \); \( \text{priv} \) is the privilege contained in \( acp \); \( \text{sbj.pk} \in \text{Sbj PK}(acp) \); \( \text{obj} \) is a document protection object of \( d \), determined according to \( \text{priv} \) and the labels in the marking containing \( acp \); \( \text{signature} \) is the digital signature of \( do \) over the certificate.

In the following, we denote with the term valid certificate a document modification certificate generated according to the policies in \( PB \) and not yet revoked. Indeed when a subject loses a right over a particular object because the \( PB \) or a credential are updated, the corresponding document modification certificates are inserted in a Revocation List that the \( do \) makes public to all the subjects involved in the cooperative update process. Moreover, the \( do \) takes also care of sending all the valid certificates to the corresponding subjects.

Example 14 Consider three users Alice, Bob and Tom with credentials company management director, manager, and secretary, respectively. Suppose moreover that Bob and Tom work in the R&D department. Consider moreover the policies in Figure 3.4 and information in Table 3.2. Then: \((4, &1, \text{update.attr}, PK_{146}, R1, \text{signature})\) is not a valid certificate, since Alice is not authorized to update attributes of region \( R1 \), but only to view their content. By contrast, \((11, &1, \text{update.attr}, PK_{102}, R3,\)
signature) and \((14, &1, update_{\text{attr}}, PK_{154}, R2, signature)\) are examples of valid certificates since Bob and Tom are authorized to update those attributes.

Similarly, a flow policy modification certificate consists of a privilege \(p\), the public key of a subject that can exercise \(p\), and the set of flow policy portions on which the subject can exercise \(p\).

We now introduce the notion of flow policy protection object, that is the formal specification of the flow policy portions on which a supported privilege can be exercised, and then we formally define a flow policy modification certificate.

**Definition 15 (Flow Policy Protection Object).** Let \(fp\) be a flow policy and let \(p\) be a privilege provided by our flow modification specification language. A flow policy protection object for \(fp\), can consist of: 1) \(r_{\text{id}}\), a flow policy region identifier belonging to \(\text{FPR}(fp)\), if \(p\) is equal to \text{update}, or 2) a root_{\text{id}} and \(\text{reg}\), where root_{\text{id}} is the flow policy atomic element root of a sub-tree in \(fp\), \(\text{reg}\) is a set of pairs \((r_{\text{id}}, \text{set}_{\text{fpae}})\), with \(r_{\text{id}}\) flow policy region identifier belonging to \(\text{FPR}(fp)\), and \(\text{set}_{\text{fpae}}\) set of flow policy atomic elements belonging to \(r_{\text{id}}\) and to the sub-tree rooted at root_{\text{id}}, if \(p\) is equal to \text{delete}.

We are now ready to formally introduce the notion of flow policy modification certificate as follows.

**Definition 16 (Flow Policy Modification Certificate).** Let \(fpo\) be a flow policy originator, let \(\mathcal{FP}(fpo)\) be the set of flow policies generated by \(fpo\) and let \(fp\) be an element belonging to \(\mathcal{FP}(fpo)\). Let \(\mathcal{R}(fp)\) be the set of flow modification rules, specified by \(fpo\), that apply to \(fp\), and let \(fmr\) be an element belonging to \(\mathcal{R}(fp)\). Let \(\text{Sbj}_{\text{PK}}(fmr)\) be the set of public keys of subjects authorized to modify \(fp\) according to \(fmr\). A flow policy modification certificate \(\text{fpmc}\), generated according to \(fmr\), is a tuple \((\text{cert}_{\text{id}}, \text{fpa}_{\text{id}}, \text{priv}, \text{sbj}_{\text{pk}}, \text{obj}, \text{signature})\), where: \(\text{cert}_{\text{id}}\) is the certificate identifier that univocally identifies a flow policy modification certificate among those generated by \(fpo\); \(\text{fpa}_{\text{id}}\) is the identifier of the flow policy attachment that contains \(fp\); \(\text{priv}\) is the privilege contained in \(fmr\); \(\text{sbj}_{\text{pk}} \in \text{Sbj}_{\text{PK}}(fmr)\); \(\text{obj}\) is a flow policy protection object of \(fp\) specified according to \(fmr\); \(\text{signature}\) is the digital signature computed by \(fpo\) on \(\text{fpmc}\).

Flow policy modification certificates containing the privilege \text{delete} allow a subject to delete all the flow policy atomic elements specified within \(\text{obj}\) component, whereas flow policy modification certificates containing the privilege \text{update} allow a subject to update one or more of the flow policy atomic elements belonging to the region specified in the \(\text{obj}\) component.
Table 6.5: Control data structures for deletable and non-deletable document atomic elements

<table>
<thead>
<tr>
<th>Name</th>
<th>Notation</th>
<th>Structure</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control structure for non-deletable document atomic elements</td>
<td>( \text{NDAE}_E \text{LIST} )</td>
<td>list of ( T_{\text{NDAE}} ), one for each non-deletable document atomic element of ( d ) belonging to a particular region ( r \text{id} )</td>
<td>Control information associated with the non-deletable document atomic elements of ( d ) belonging to a particular region ( r \text{id} )</td>
</tr>
<tr>
<td>Control tuple for non-deletable document atomic element</td>
<td>( T_{\text{NDAE}} )</td>
<td>((\text{docae}_E \text{id}, \text{position}, \text{data}))</td>
<td>Information corresponding to a non deletable atomic element ( \text{docae}_E ) of a document ( d )</td>
</tr>
<tr>
<td>Control structure for deletable document atomic elements</td>
<td>( \text{DAE}_E \text{LIST} )</td>
<td>list of ( T_{\text{DAE}} ), one for each deletable document atomic element of ( d ) belonging to a particular region ( r \text{id} )</td>
<td>Control information associated with the deletable document atomic elements of ( d ) belonging to a particular region ( r \text{id} )</td>
</tr>
<tr>
<td>Control tuple for deletable document atomic element</td>
<td>( T_{\text{DAE}} )</td>
<td>((\text{docae}<em>E \text{id}, \text{position}, \text{data}, h</em>{\text{docae}}))</td>
<td>Information corresponding to a deletable document atomic element ( \text{docae}_E ) of a document ( d )</td>
</tr>
</tbody>
</table>

**Example 15** Consider user Bob, with credential manager. Consider moreover the atomic flow policy attachment constituted by the portions in Figures 6.1, the policies in Figure 6.2 and information in Table 6.4. Then: \((10, \text{F1, delete, PK}_{154}, 12.\text{tags\&attrs}, \{(\text{Reg}3, \{12.\text{tags\&attrs, 13.\text{tags\&attrs, 14.\text{tags\&attrs, 15.\text{tags\&attrs}}\}, (\text{Reg}2, \{14.\text{dc, 15.\text{dc}}\}), \text{signature}\) is a valid certificate, since Bob is authorized to delete the sub-tree rooted at the ReceiverSpec with identifier equal to 12. By contrast \((14, \text{F1, update, PK}_{102}, \text{Reg}2, \text{signature}\) is not a valid certificate for Bob since Bob is not authorized to update CredSpec elements with identifier equal to \((3\vert 6\vert 10)\) and ExtSpec elements with identifier equal to \((4\vert 7\vert 11)\).

As already said in Chapter 3 our framework supports two modes (on-line and off-line) for the distribution of the information needed by subjects to accomplish their job during the update process.
Table 6.6: Components of the control data structures for deletable and non-deletable document atomic elements

<table>
<thead>
<tr>
<th>Component</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>docae_id</td>
<td>identifier of the document atomic element docae</td>
</tr>
<tr>
<td>position</td>
<td>value that specifies where docae’s components are located in the original document d</td>
</tr>
<tr>
<td>data</td>
<td>encrypted docae’s content</td>
</tr>
<tr>
<td>h_docae</td>
<td>hash value computed over the data component</td>
</tr>
</tbody>
</table>

6.4.2 Document Control Information

The approach presented in this chapter requires a control information portion to be computed on the document at the beginning of the update process by do and other document control information portions to be updated during the process itself to keep track of the modification to the document content. As already seen each document atomic element is marked with a label containing the set of access control policies that apply to it. We can distinguish two main categories of document atomic elements, according to the privileges contained in those policies: non-deletable atomic elements and deletable atomic elements. We split document atomic elements in two categories according to the possibility to be deleted, because a deletable element requires the computation of additional control information wrt a non-deletable one. In this way we can minimize the amount of control information to be computed and inserted in the document package. Examples of deletable atomic elements are attributes to which at least an access control policy with the delete_attr privilege applies or attributes and tags to which at least an access control policy with a delete_elem privilege applies. Table 6.5 shows control data structures associated with both the categories, whereas Table 6.6 presents the components of the structures introduced in Table 6.5.

Document regions generated by the marking of the document can be divided in non-modifiable and modifiable regions. A region is non-modifiable if all policies that apply to it contain only browsing privileges; a region is modifiable otherwise. Table 6.7 presents the control data structures for non-modifiable regions, whereas Table 6.8 illustrates the semantics of components presented in Table 6.7. We use character (∗) to denote the string concatenation operator, and we use the notation (∑∗x∈ListX x) to denote the concatenation of all the elements belonging to ListX, in the order in which they are listed.

Modifiable regions can be further classified into five sub-categories, according to the different authoring privileges contained in the access control policies that apply to them. This dis-
Table 6.7: Control data structures for non-modifiable document regions

<table>
<thead>
<tr>
<th>Name</th>
<th>Notation</th>
<th>Structure</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control structure for non-modifiable document regions</td>
<td>( NMR )</td>
<td>list of ( T_{NMR} ), one for each non-modifiable document region of ( d )</td>
<td>Information used by a subject to verify integrity of non-modifiable document regions of ( d )</td>
</tr>
<tr>
<td>Control tuple for non-modifiable document regions</td>
<td>( T_{NMR} )</td>
<td>((r_{id}, NDAE_LIST, h_{nmr_static}))</td>
<td>Information corresponding to a specific non-modifiable document region ( r_{id} ) of ( d )</td>
</tr>
</tbody>
</table>

Table 6.8: Components of the control data structures for non-modifiable document regions

<table>
<thead>
<tr>
<th>Component</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_{id} )</td>
<td>identifier of a non-modifiable document region of a document ( d )</td>
</tr>
<tr>
<td>( h_{nmr_static} )</td>
<td>hash value computed over all the components of the elements in ( NDAE_LIST ) belonging to ( r_{id} ): ( \text{H}(\sum_{t \in \sum_{T_{NMR}(r_{id}), NDAE_LIST} \text{docae}_{id} \ast t.\text{position} \ast t.\text{data}) )</td>
</tr>
</tbody>
</table>

Table 6.9 presents these five sub-categories, giving, for each sub-category, the corresponding authoring privileges. For example the set of authoring privileges contained in the access control policies that apply to a region classified as \( PDUR \) must be equal to the set \( \{ \text{update}\_\text{attr}, \text{delete}\_\text{attr} \} \).

Without lack of generality in the following we focus only on fully deletable and updatable regions \( (FDUR) \), because they are the modifiable regions on which all the authoring privileges supported by our model can be exercised and thus they represent the most complex modifiable region sub-category. According to this assumption Table 6.10 presents the control data structure for \( FDUR \) regions. In the same way, Table 6.11 presents the components of the control data structure for \( FDUR \) regions, where \( \text{delete\_elm\_cert} \) contains the certificates with \( \text{delete\_elm} \) privilege, inserted by subjects when they exercised their modification rights, that apply to disjoint set of atomic elements, thus defined non-overlapping certificates.

The signature generated by the last subject that has modified the content of a modifiable
<table>
<thead>
<tr>
<th>Sub-category</th>
<th>Notation</th>
<th>Privileges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Updatable regions</td>
<td>UR</td>
<td>{update_attr}</td>
</tr>
<tr>
<td>Partially deletable regions</td>
<td>PDR</td>
<td>{delete_attr}</td>
</tr>
<tr>
<td>Fully deletable regions</td>
<td>FDR</td>
<td>{delete_elmt} or {delete_elmt delete_attr}</td>
</tr>
<tr>
<td>Partially deletable and updatable regions</td>
<td>PDUR</td>
<td>{update_attr, delete_attr}</td>
</tr>
<tr>
<td>Fully deletable and updatable regions</td>
<td>FDUR</td>
<td>{update_attr, delete_elmt} or {update_attr, delete_elmt delete_attr}</td>
</tr>
</tbody>
</table>

Table 6.9: Modifiable Region Classification

<table>
<thead>
<tr>
<th>Name</th>
<th>Notation</th>
<th>Structure</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control structure for document modifiable regions</td>
<td>DMR</td>
<td>(UR, PDR, FDR, PDUR, FDUR, delete_elmt_cert)</td>
<td>Information used by a subject to verify correctness and integrity of document modifiable regions</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Control structure for fully deletable and updatable regions</td>
<td>FDUR</td>
<td>list of T_{FDUR}, one for each fully deletable and updatable region of d</td>
<td>Information used by a subject to verify integrity of FDUR regions of d</td>
</tr>
<tr>
<td>Control tuple for fully deletable and updatable regions</td>
<td>T_{FDUR}</td>
<td>(r_{id}, DAE_LIST, h_{fdur static}, sig_{fdudocae}, update_cert, delete_attr_cert)</td>
<td>Information corresponding to a specific FDUR deletable and updatable region r_{id}</td>
</tr>
</tbody>
</table>

Table 6.10: Control data structures for modifiable document regions

region is computed on the components h_{docae} associated with the atomic elements belonging to that region and not on their content (component data). This is done to allow the integrity check of the region after the computation of the signature, also in presence of deletion of some atomic elements belonging to the region. Indeed it will be enough to check the integrity of the components h_{docae} and then to check the correspondence between each h_{docae} component and the corresponding atomic element content (component data), only for those elements non declared as deleted. The same must be done for modification operations executed on the flow policy.
Component | Meaning and formal specification
--- | ---
delete\_elm\_cert \( \) it contains non-overlapping authoring certificates, with privilege equal to delete\_elm, inserted by the subjects that have executed a deletion over one or more document regions
\( r\_jd \) \( \) identifier of a modifiable document region of a document \( d \)
h\_fdur\_static \( \) hash value computed by do over docae\_id, position and h\_docae components of the document atomic elements that are tags and also over docae\_id and position components of the document atomic elements that are attributes listed in DAE\_LIST belonging to a FDUR \( r\_jd \):

\[
H( \sum_{t \in \text{FDUR}[r\_jd].\text{DAE\_LIST}.\text{type}(t) = \text{tags}} t.docae\_id \ast t.position \\
\ast \sum_{t \in \text{FDUR}[r\_jd].\text{DAE\_LIST}.\text{type}(t) = \text{attributes}} t.docae\_id \\
\ast t.position )
\]
\( \) \( \)

sig\_fdudocae \( \) digital signature computed over the h\_docae component of all the document atomic elements that are attributes listed in DAE\_LIST belonging to a FDUR \( r\_jd \) by the last subject \( (S_{\text{last}}) \) that has modified the region and whose modification declaration is contained in the receiver specification identified by the information: \( (\text{fpa}-\text{id}, \text{ver}, r\_id, \text{orig}) \), where fpa\_id is a fpa identifier, ver is a fpa version, rs\_id is a receiver specification identifier and orig is a fpa originator

\[
S_{\text{last}}( \sum_{t \in \text{FDUR[r\_jd].DAE\_LIST.type}(t) = \text{attribute}} t.h\_docae) \ast \text{fpa}\_id \\
\ast \text{ver} \ast \text{rs\_id} \ast \text{orig }
\]
update\_cert \( \) it contains the authoring certificate with privilege equal to update\_attr inserted in a region \( r\_jd \) by the last subject \( (S_{\text{last}}) \) that has updated that region
\( \) \( \)
delete\_attr\_cert \( \) it contains the authoring certificate with privilege equal to delete\_attr inserted in a region \( r\_jd \) by the last subject \( (S_{\text{last}}) \) that has deleted at least one attribute of that region

Table 6.11: Components of the control data structures for FDUR

6.4.3 Modification declarations

The update of the document and flow policy content requires the insertion in the flow policy attachment of some modification declarations. A declaration can describe an operation executed on the document/flow policy atomic elements belonging to a single modifiable region (update\_attr/delete\_attr access control policy privileges or update flow modification privilege) or belonging to more than one modifiable region (delete\_elem access control policy privilege or delete flow modification privilege).

Modification declarations are used, for example, by a subject to check the document in-
Chapter 6: Cooperative updates in colluding byzantine and failure prone distributed systems

Notation | Structure | Semantics
---|---|---
ReceiverSpec | \(\ldots, \text{DocDecl}, \text{FpDecl}, \ldots\) | Single receiver specification associated with information inserted by the corresponding receiver

DocDecl | \(\text{Doc-UpAttr-Decl, Doc-DelAttr-Decl, Doc-DelEl-Decl}\) | Modification declaration inserted by a receiver when it modifies the document content

Doc-UpAttr-Decl | Set of \(r_{id}\) | Set of document regions declared as updated by the receiver

Doc-DelAttr-Decl | Set of \((r_{id}, \text{del-docae})\) | Declaration inserted by the receiver when it deletes some attributes belonging to the document

del-docae | Set of \(docae_{id}\) | Set of document atomic elements (attributes) declared as deleted by the receiver

Doc-DelEl-Decl | Set of \((doc-root_{id}, del-reg)\) | Deletion declaration concerning some sub-trees of the document

del-reg | Set of \(r_{id}\) | Set of regions involved in a deletion of a document sub-tree

FpDecl | \(\text{set of } (fpa-id, fpa-version, fpa-orig, Fp-Up-Decl, Fp-Del-Decl)\) | Modification declaration inserted by a receiver when it modifies the content of one or more flow policies

Fp-Up-Decl | Set of \(r_{id}\) | Set of flow policy regions declared as updated by the receiver

Fp-Del-Decl | Set of \(fp-root_{id}\) | Deletion declaration concerning some sub-trees of the flow policy

Table 6.12: Modification declaration structure within a receiver specification

tegrity, verifying that each current region content is correct with respect to the signature computed by the last subject that has declared the exercise of update\(_{attr}\) on the region and the existence of the corresponding document modification certificate. Moreover, in presence of delete\(_{attr}\) declarations regarding a modifiable region, then must exist a certificate corresponding to the last subject that has declared the exercise of delete\(_{attr}\) privilege on that region.

The flow policy attachment provides different ad-hoc control structures to store modification declarations concerning the document content and others concerning flow policy content. Table 6.12 presents this control information, whereas Table 6.13 presents components of the structure for modification declarations. At the end of the document and flow policy content update, must
Component | Semantics
--- | ---
r\_id | document/flow policy region identifier
docae\_id | document atomic element identifier corresponding to an attribute of the document
doc-root\_id | document atomic element identifier corresponding to the root of the sub-tree declared as deleted
fpa-id | flow policy attachment identifier
fpa-version | flow policy attachment version
orig | flow policy attachment originator
fp-root\_id | flow policy atomic element identifier corresponding to the root of the sub-tree declared as deleted

Table 6.13: Components of the Modification Declaration Structure

insert, for each modification operation executed on the document, a declaration that describes that operation, a document modification certificate, guaranteeing subsequent receivers that \( s \) possesses the privilege required to execute that operation, and, in case the privilege is update\_attr, it must compute a new signature on the updated content. Document modification certificates are inserted: in the unique component delete\_elm\_cert, if they contain the delete\_elm privilege; in the component update\_cert associated with the modified region, if they contain update\_attr privilege, whereas those containing delete\_attr are inserted in the component delete\_attr\_cert associated with the modified region. A similar strategy must be followed when \( s \) inserts a declaration for a modification operation executed on a flow policy.

### 6.5 Distributed and cooperative update process

Our approach relies on a suite composed of different protocols. More precisely, the suite consists of the protocol executed by the do (Document Originator Protocol), the one executed by the subjects in CG (Subject Protocol), the one executed by the operative delegates (Delegate Protocol), and finally, that executed by the down delegates (Down Delegate Protocol). Figures 6.5, 6.6, 6.7, and 6.8 present all of these protocols, whereas Tables 6.18 and 6.19 present functions invoked in the protocols.

In what follows we describe phases and corresponding sub-phases that occur during the distributed and cooperative update process, referring, in this presentation, the involved protocols. Before doing that we discuss some properties required by the protocols, the assumptions over which
our approach relies and how are set the parameters required by the protocols.

6.5.1 Assumptions and Properties

Our approach relies on a set of assumptions and needs that some properties hold to correctly manage a cooperative and distributed update process. First, we assume that the do, each delegate and each subject involved in the update process possesses a private key and all the other parties know or can collect the public key of each other. The do is in charge of informing, at the beginning, all subjects and delegates of which users compose CG and DG, since this information is required by the Subject Protocol and by the Delegate and Down Delegate Protocols. Moreover, we assume that communication is reliable and has a finite upper bound on message transmission time. This means that if an honest party sends a message to another honest and reachable party, the message is received by a fixed amount of time (MTTIME). Each sent message is always signed by the sender for integrity and authentication purposes. To avoid deadlocks caused by the malicious behaviour of a byzantine sc, after a fixed amount of time between two change of state executed by an operative delegate a Rollback procedure starts to replace sc with another subject. In the protocols specified in this chapter we do not treat this procedure and we assume that a sc does not retain the document an undefined amount of time and that it submits to delegates a correct updated Fpa for the integrity check. We call State\textsuperscript{x}dl the state associated with a delegate dl ∈ OP at step x. State\textsuperscript{x}dl contains different components that can be possibly updated step by step: a Document (Doc), a Flow policy attachment (Fpa), a structure containing the invalid modification document declarations (IMDD), a structure used during the Recovery sub-phase that indicates when the last recovery occurred for a region (LSRR), a vector of progressive numbers used to avoid reply attacks (NI\textsubscript{TP}, where TP = CG ∪ DG ∪ {do}) and a set components that do not change during the update process: the Delegates Group (DG) and the cooperative Group (CG). In particular, during a recovery of the document content each time a declaration is found invalid, because the subject that has generated that declaration is unreachable, it does not possess the corresponding correct document modification certificate or it incorrectly exercised the corresponding privilege, that declaration is inserted in a particular structure, IMDD (Invalid Modification Document Declarations), presented in Table 6.14. Invalid declarations are stored in IMDD according to the subject that has inserted them in Fpa, and according to the type of operation associated with it (update\textsubscript{e} attr/delete\textsubscript{e} attr/delete\textsubscript{e} elmt privilege). A subject is identified, in IMDD, through the information that specifies its position in Fpa at the time of insertion of that declaration in Fpa itself. This structure is received by sc at
Table 6.14: Structure of the Invalid Modification Document Declaration information

<table>
<thead>
<tr>
<th>Notation</th>
<th>Structure</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMDD</td>
<td>(doc-id, doc-version, doc-orig, sbj-inv-decl)</td>
<td>structure containing all the originator declarations that invalidate subject declarations inserted in the flow policy</td>
</tr>
<tr>
<td>sbj-inv-decl</td>
<td>set of (fpa-id, fpa-version, fpa-orig, rs-id, Up-Attr, Del-Attr, Del-Elmt)</td>
<td>set of no more valid declarations</td>
</tr>
<tr>
<td>Up-Attr</td>
<td>set of r-id</td>
<td>invalid declaration concerning an update operation exercised over region r-id</td>
</tr>
<tr>
<td>Del-Attr</td>
<td>set of r-id</td>
<td>invalid declaration concerning a delete attribute operation exercised over region r-id</td>
</tr>
<tr>
<td>Del-Elmt</td>
<td>set of doc-root-id</td>
<td>invalid declaration concerning a delete operation exercised over the sub-tree with root doc-root-id</td>
</tr>
</tbody>
</table>

the beginning and used to check the integrity of document content. Indeed, document content must result correct wrt only valid modification declarations, that is modification declarations stored in the received Fpa and that are not present in the received IMDD structure. During a recovery, a further data structure is used, called LSRR (Last Saved Region Recovery). This structure stores, for each modifiable region, the information that identifies the subject in Fpa that has generated the last detected as corrupted version of the document wrt that region. This is done because during a recovery only subjects, that has declared some modifications on a region to be recovered, and that appear in Fpa in a position greater than that stored in LSRR for that region, will be contacted to obtain the most recent correct region content. Since previous recovery has stored the most recent correct region content wrt the declarations in Fpa inserted by subjects in a position less than that stored in LSRR for that region, the recovery process will use it to recover the region if it does not receive a more recent correct region content by the contacted subjects.

For a delegate $dl \in OP$, a step $x$ ends and the subsequent one $(x + 1)$ begins when $dl$ makes stable $State_{dl}^{x+1}$, that is, the values of modifiable information contained in $State_{dl}^{x}$ are replaced with the new ones, according to the information contained in the last correct message sent by $s_c$ to all delegates. This message is considered correct by the operative delegates if the contained information is signed by a number of delegates denoted, here and in what follows, as Quorum.
Chapter 6: Cooperative updates in colluding byzantine and failure prone distributed systems

<table>
<thead>
<tr>
<th>Notation</th>
<th>Structure</th>
<th>Protocols</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>variable containing a string</td>
<td>Delegate/Subject</td>
<td>it indicates in which state a delegate/subject is in a particular moment during the update process</td>
</tr>
<tr>
<td>requests</td>
<td>variable containing an integer</td>
<td>Delegate</td>
<td>contains the number of processed requests wrt the value of variable state</td>
</tr>
<tr>
<td>N_{IP}</td>
<td>vector of integers</td>
<td>Delegate</td>
<td>it is used to keep track of the number of steps a subject or do have taken part, and how many times a delegate has requested an agreement</td>
</tr>
<tr>
<td>Queue</td>
<td>buffer containing the received messages</td>
<td>Delegate</td>
<td>it collects all the messages received from the various parties</td>
</tr>
<tr>
<td>N_{CG}</td>
<td>vector of integers</td>
<td>Subject</td>
<td>it is used to keep track of the number of steps s_c itself or another subject in CG have taken part</td>
</tr>
</tbody>
</table>

Table 6.15: Data structures used by the protocols

(Q). Protocols assure that each delegate signs only once such an information at each step, thus preventing byzantines from the possibility of obtaining two different contents signed by at least Q distinct delegates and forcing different delegates, such as dl_i, dl_j ∈ OP, to make stable State_{dl_i}^{x+1} and State_{dl_j}^{x+1} with State_{dl_i}^{x+1} ≠ State_{dl_j}^{x+1}. Thus we can use State^x to indicate the common state of all operative delegates for each step x, x ≥ 0. We also call sub-step all the statements executed by a party when a particular event occurs within a step. In Table 6.15 we presents all the data structures used by the protocols, that we describe in more detail in what follows.

More precisely, the Delegate Protocol makes use of some variables and structures. In particular, a state that indicates the state in which the delegate is (for example, norec indicates that the delegate has not yet requested a recovery or it is completing a step without the need of a recovery). By contrast, variable requests contains the number of processed requests wrt the value of variable state. Indeed, if no recovery has been requested, variable requests reaches at most value 1, whereas it reaches value 2 in presence of recovery. These variables are used to avoid reply attacks within the same step. Structure N_{IP} is used to keep track of the number of steps a subject or the do has taken part, and how many times a delegate has requested an agreement. This structure is a vector of progressive numbers, initially sets to all zeros, one for each party involved in the process. Whenever a subject/do participates in a step, the corresponding progressive number is incremented. The indication of the receiver, in the messages sent by a delegate, together with the insertion of the value stored in N_{IP}, corresponding to that receiver, prevents byzantine subjects and/or byzantine
delegates from replying messages exchanged in a step \( x \) during a step \( y \), with \( y > x \). Another used data structure is \textit{Queue}, that stores all the received messages. A delegate, during a generic step \( x \), needs a strategy to choose among all the received messages the next one to be processed. This strategy is called \textit{received messages scheduling policy} and it is applied each time a message has been completely processed or when the time assigned to a process that processes a message ends. This policy collects among all the messages in \textit{Queue} only the messages valid according to \textit{State}, and the values of the previous introduced variables. Then, it selects from this set the messages with higher priority and finally, if it is found more than one message, the message received first. Subject Protocol makes use of variable \textit{state}, that represents the action the subject is executing or has been just executed. Structure \( NC\) used by the subjects is similar to \( NIP\), but keeps track of the number of steps the subject itself and the other subjects have taken part. Whenever a subject \( s \) sends a message, this message contains the progressive number associated with \( s \) (\( NG[s] \)). This information is used by a receiver to discard old messages.

### 6.5.2 Protocol Parameters Setting

At the beginning of the update process the \textit{do} has to set two parameters: \( b \) and \( d \). The system proposes for parameter \( b \) a default value equal to 0, that can be changed by the \textit{do}; by contrast for the parameter \( d \) the system proposes an estimated average number of failures (\( f \)), determined monitoring the network, giving the possibility to the \textit{do} to set a correction parameter (\( c \)), whose default value is 1, that multiplied by \( f \) gives the value (\\ceil{c \cdot f}) to be assigned to parameter \( d \). According to these considerations we can formally define cardinality of \( DG \) as follows.

\textbf{Definition 17 (Delegates Group cardinality)} Let \( b \) and \( d \) be the parameters, set by the \textit{do} at the beginning of the update process, that respectively represent the maximum number of byzantine delegates that do not affect the protocol, and the maximum number of down delegates that do not delay the protocol. Let also \( op \) be the number of operative delegates, reached when there are exactly \( b \) byzantine and \( d \) down delegates and such that it does not delay the protocol. Cardinality of \( DG \) can be defined as follows: \( DG = b + op + d \).

At this point we have to determine the value to be associated with parameter \( op \), to obtain the exact cardinality of \( DG \). Value of parameter \( op \) is strictly related to \( Q \). Indeed \( op \) must be greater than or equal to \( Q \), because in case byzantines delegates do not answer a request, only operative delegates will be able to sign a message content for which the protocol requires at least
Figure 6.3: Message exchange

\( \mathcal{Q} \) valid signatures. Moreover, byzantines delegates do not have to be able to obtain two sets of valid signatures of cardinality at least equal to \( \mathcal{Q} \) for two different messages of the same type, or for two messages of the same priority, exchanged in the same step, in the case in which there is the maximum number of byzantines delegates and no down delegate.\(^4\)

Formally the correct value of \( \mathcal{Q} \) that assures all the above requirements, and the corresponding value for \( \mathcal{Q} \) are as follows.

**Proposition 6** (\( \mathcal{Q} \) and \( \mathcal{Q} \) values). Minimum value of \( \mathcal{Q} \) such that:

a) \( \mathcal{Q} \) is greater than or equal to \( \mathcal{Q} \);

b) byzantines delegates cannot obtain two sets of valid signatures of cardinality greater than or equal to \( \mathcal{Q} \) for two different contents contained in two messages of the same type, or for two messages of the same priority, exchanged in the same step, in the worst case in which there is the maximum number of byzantines delegates and no down delegate.\(^4\)

is equal to: \( 2b + d + 1 \). Corresponding \( \mathcal{Q} \) is equal to: \( 2b + d + 1 \).

\(^{4}\)More details about message types/priorities and number of required signatures for a message type are presented in Section 6.5.3.
6.5.3 Update process phases

Parties involved in a cooperative and distributed update process communicate exchanging messages of different type and content according to the protocol executed by each party. Figure 6.3 shows an overall picture of the messages that are exchanged by the involved parties, whereas Tables 6.16 and 6.17 gives more details about the content of the exchanged messages. More precisely messages are presented in terms of associated priority (P), type, sender, receiver(s), and giving its complete structure and semantics. Only messages received by a delegate have associated a priority. This is done because only Down Delegate and Delegate Protocols use this information to choose the next message to be processed.

We can indicate three main phases that occur during an update process: Initialization phase, cooperative update phase, Final phase. In particular the second one consist of some sub-phases. In the following we present these three main phases and subsequently the sub-phases required by the protocols.

6.5.3.1 Initialization phase

The do chooses CG and DG (line 6.5.10), distributes to all delegates \((Doc_{do}, Fpa_{do}, CG, DG)\) (line 6.5.12). In Document Originator Protocol we specify the statements that realize the on-line distribution mode of decryption keys and document/flow policy modification certificates (lines 6.5.13-15) Finally, the do sends the first subject the do’s version of the document to be updated \((Doc_{do})\) and the do’s version of the associated flow policy attachment \((Fpa_{do})\) (line 6.5.16).

6.5.3.2 Cooperative update phase

\(s_c\) receives from the previous current subject \((s_{pc})\) the last version of document \(Doc\) \((s_{pc}’s\ document version)\) and from the delegates the last updated and correct \(Fpa\) associated with \(Doc\) in a step \(x, x > 0\) (line 6.6.07). If \(s_c\) is the first receiver it receives \(Doc\) and \(Fpa\) directly from the do (line 6.6.05) in step 0, and thus no check is required over the document content, and \(s_c\) can immediately exercise its modification rights on the document content. Otherwise, upon receiving \(Doc\) and \(Fpa\), \(s_c\) checks the integrity of non-modifiable document information and of non-modifiable regions that it can access. Then, it checks content integrity of modifiable regions that it can access, and if an error occurs it starts a Recovery sub-phase. During this sub-phase delegates replace non-modifiable information within the corrupted received document with that contained in \(Doc\) belonging to \(State^2\). Then they contact subjects in \(CG\) to obtain the last correct content of modi-
### Table 6.16: Messages (part 1)

<table>
<thead>
<tr>
<th>P</th>
<th>Type</th>
<th>Sender</th>
<th>Recvr(s)</th>
<th>Content and Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>init-dg</td>
<td>do</td>
<td>DG</td>
<td>(init-dg, d\text{ld}, Doc_{do}, Fpa_{do}, CG, DG)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Message sent by the do (do) to all delegates containing the original version of the document, the initial flow policy attachment and the set of delegates and subjects involved in the process</td>
</tr>
<tr>
<td>-</td>
<td>init-cg</td>
<td>do</td>
<td>CG</td>
<td>(init-cg, d\text{ld}, regkeys_s, docmodcert_s, fpmodcert_s, CG, DG)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Message sent by the do to each subject containing s’s decryption keys, document/flow policy certificates and the set of subjects and delegates</td>
</tr>
<tr>
<td>0</td>
<td>agreement</td>
<td>dl ∈ D</td>
<td>DG</td>
<td>(agreement)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Message sent by a down delegate to all delegates in DG to receive information needed to reach the same state of the operative delegates</td>
</tr>
<tr>
<td>-</td>
<td>agreement-resp</td>
<td>DG</td>
<td>dl ∈ D</td>
<td>(agreement-resp, history, hpm, agreements, dl, N_EP[dl])</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Message sent by delegates to a down delegate containing the history of all previous steps, in terms of messages of type fw-signed-fpa-nr or fw-signed-fpa-ar that cause the step change (history), their last processed message (hpm), all the received but not still processed agreement messages (agreements) and information required to prevent other delegates to reply this message (a progressive number and the public key of the down delegate receiver)</td>
</tr>
<tr>
<td>1</td>
<td>err</td>
<td>s_c ∈ CG</td>
<td>do  DG</td>
<td>(err, m, S_{sbj}(m), MReg, N_{CG}[s_c])</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Message sent by the current subject/do to all delegates when an error occurs to the document content to collect ((b + 1)) recovery versions</td>
</tr>
<tr>
<td>-</td>
<td>rec</td>
<td>DG</td>
<td>s_c ∈ CG</td>
<td>do  (rec, IMDD_{dl}, MReg, Doc_{dl}, sbj, n_{sbj})</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Message sent by delegates to the current subject/do containing the result of their recovery: a Doc and the updated IMDD structure</td>
</tr>
<tr>
<td>2</td>
<td>fw-rec</td>
<td>s_c ∈ CG</td>
<td>DG</td>
<td>(fw-rec, {m, S_{dl}(m)}_{dl\in D})</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Message sent by the current subject to all delegates to receive the last correct document version wrt its accessible modifiable regions, obtained unifying the (</td>
</tr>
<tr>
<td>-</td>
<td>rec-merge</td>
<td>DG</td>
<td>s_c ∈ CG</td>
<td>(rec-merge, IMDD_{dl}, MReg, Doc_{dl}, sbj, n_{sbj})</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Message sent by delegates to the current subject containing a Doc, union of the ((b + 1)) received recoveries, and the IMDD structure, updated according to the ((b + 1)) received recoveries</td>
</tr>
</tbody>
</table>
Table 6.17: Messages (part 2)

<table>
<thead>
<tr>
<th>P</th>
<th>Type</th>
<th>Sender</th>
<th>Recvr(s)</th>
<th>Content and Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>new-fpa-nr</td>
<td>$s_c \in CG$</td>
<td>$DG$</td>
<td>$(new\text{-}fpa\text{-}nr,Fpa_{s_c})$ Message sent by the current subject to all delegates to propose a new $Fpa$ to be made stable, in absence of recovery</td>
</tr>
<tr>
<td></td>
<td>signed-fpa-nr</td>
<td>$DG$</td>
<td>$s_c \in CG$</td>
<td>$(signed\text{-}fpa\text{-}nr,Fpa)$ Message sent by delegates to the current subject if the proposed $Fpa$ is correct, in absence of recovery</td>
</tr>
<tr>
<td>3</td>
<td>new-fpa-ar</td>
<td>$s_c \in CG$</td>
<td>$DG$</td>
<td>$(new\text{-}fpa\text{-}ar,Fpa_{s_c})$ Message sent by the current subject to all delegates to propose a new $Fpa$ to be made stable, after a recovery</td>
</tr>
<tr>
<td></td>
<td>signed-fpa-ar</td>
<td>$DG$</td>
<td>$s_c \in CG$</td>
<td>$(signed\text{-}fpa\text{-}ar,Fpa)$ Message sent by delegates to the current subject if the proposed $Fpa$ is correct, after a recovery</td>
</tr>
<tr>
<td>4</td>
<td>fw-signed-fpa-nr</td>
<td>$s_c \in CG \mid DG$</td>
<td>$DG$</td>
<td>$(fw\text{-}signed\text{-}fpa\text{-}nr, m, {S_d(m)}_{d \in Q})$ Message sent by the current subject to all delegates and then forwarded by delegates to each other delegate to make stable the $Fpa$ contained in $m$ and previously signed by $</td>
</tr>
<tr>
<td></td>
<td>fw-signed-fpa-ar</td>
<td>$s_c \in CG \mid DG$</td>
<td>$DG$</td>
<td>$(fw\text{-}signed\text{-}fpa\text{-}ar, \overline{m}, {S_d(\overline{m})}<em>{d \in \overline{Q}}, \overline{m}, {S_d(\overline{m})}</em>{d \in \overline{Q}})$ Message sent by the current subject to all delegates and then forwarded by delegates to each other delegate to make stable the Doc contained in $\overline{m}$ and the $Fpa$ contained in $\overline{m}$ and previously signed by $</td>
</tr>
<tr>
<td>5</td>
<td>end</td>
<td>do</td>
<td>$DG \cup CG$</td>
<td>$(end, d_{id})$ Message sent by the do to all delegates and subjects to end the update process</td>
</tr>
</tbody>
</table>

fiable regions belonging to the set of regions indicated by $s_c$ in its recovery request, and after that they send their recovered document and corresponding updated IMDD structure to $s_c$. $s_c$ collects the first $(b + 1)$ responses and then sends them to the delegates, that return a document and an IMDD structure obtained composing the received responses. $s_c$ collects the first $Q$ messages with the same content, containing a recovered document and an IMDD structure, and received exactly from $Q$ distinct delegates. This method assures that the recovered document and IMDD structure are obtained starting from at least a recovery version generated by an operative delegate (due to the collection of $(b + 1)$ recovery versions), and since the composition of the $b + 1$ recovery versions is the same for $Q$ delegates, it was surely generated according to the protocol. Moreover, if $s_c$ has received at least $Q$ responses containing the same recovered document and IMDD structure,
then $s_c$ cannot collect at least $Q$ responses for another recovered version in the same step. These properties will be formally stated at the end of this section and proved in Appendix A. At this point $s_c$ is able to update the document according to its authorizations. Then, $s_c$ chooses $s_{\text{next}}$, updates, according to its authorizations, $Fpa$ content and then it starts a $Fpa$-Checking sub-phase. In this sub-phase $s_c$ sends its updated $Fpa$ to the delegates to be checked. If the check is successfully completed, delegates return a response containing the proposed $Fpa$. They send no response, otherwise, and they will check no other $Fpa$. Then, $s_c$ starts a Change-Delegates-State sub-phase in which it notifies all delegates the document and the $IMDD$ structure received after the recovery, if any, and the previously proposed correct $Fpa$. Each delegate $dl$ according to the received information changes modifiable information contained in $State_{dl}$, generating $State_{dl}^{x+1}$. Since in this phase a delegate can accept two types of message: the first containing only the new $Fpa$, whereas the second containing both a document, structure $IMDD$ and the new $Fpa$, is required that different messages must be sent for the Fpa-checking sub-phase, to prevent a byzantine subject from sending to a subset of $DG(DG_i)$ a message of the first type signed by at least $Q$ delegates and to another subset of $DG(DG_j)$, a message of the second type, signed by at least $Q$ delegates, including the recovered document, structure $IMDD$ and the same $Fpa$, inducing two delegates $dl_i \in DG_i$ and $dl_j \in DG_j$ to generate $State_{dl_i}^{x+1}$ and $State_{dl_j}^{x+1}$, with $State_{dl_i}^{x+1} \neq State_{dl_j}^{x+1}$. Finally, $s_c$ sends its document version to $s_{\text{next}}$. If $s_c$ is the last subject that has to receive the document, it sends its updated document to the $do$.

Figure 6.4 shows the message exchange between the involved parties for each sub-phase presented in what follows.

- **Recovery sub-phase:** During a Recovery sub-phase $s_c$, that has received a message $m$ from a subject $sbj$ containing a corrupted document, sends to all delegates a message $(err, m, S_{sbj}(m), MReg, NCG[s_c])$ (line 6.6.12) to obtain the last correct document version wrt the set of modifiable regions it can access ($MReg$), that is correct content of all non-modifiable document information/regions and of modifiable regions in $MReg$. As will be also presented in the final phase, each delegate $dl \in DG$ generates a document recovery version and the corresponding $IMDD$ updated structure, contacting subjects in $CG$ and then sends a message $(rec, IMDD_{dl}, MReg, Doc_{dl}, s_c, n_{s_c})$ (line 6.7.15), with $(n_{s_c} = NCG[s_c])$, containing the generated information, to $s_c$. $s_c$ accepts the first $(b + 1)$ messages from the delegates (line 6.6.14) and then forwards them in message $(fw-rec, \{m, S_d(m)\}_{dl \in D})$, with $|D| = b + 1$, to all delegates (line 6.6.16). Each delegate $dl \in DG$ composes the received $(b + 1)$ document
Chapter 6: Cooperative updates in colluding byzantine and failure prone distributed systems

Figure 6.4: Sub-phases messages exchange
Chapter 6: Cooperative updates in colluding byzantine and failure prone distributed systems

(6.5.01) state ← initial
(6.5.02) Let $Doc_{do}$ be the document to be updated and $Fpa_{do}$, the associated flow policy attachment both chosen by the do
(6.5.03) Let $NMReg$ and $MReg$ be the set of non-modifiable/modifiable regions associated with $Doc_{do}$
(6.5.04) Let $d_{id}$ be the identifier of $Doc_{do}$
(6.5.05) repeat
(6.5.06) [] state = initial:
(6.5.07) Generation of the package to be sent to the first chosen subject, $P ← (q, Doc_{do}, Fpa_{do})$
(6.5.08) Set parameters $b$ and $d$ to the value chosen by the document originator (do)
(6.5.09) Choice of subjects in $CG$ and delegates in $DG$.
(6.5.10) $m ← (init-dg, d_{id}, Doc_{do}, Fpa_{do}, CG, DG)$
(6.5.11) send $(m, S_{do}(m))$ to $DG$
(6.5.12) For each $s ∈ CG$:
(6.5.13) $m ← (init-cg, d_{id}, regkeys_s, docmodcerts_s, fpmodcerts_s, CG, DG, s)$
(6.5.14) send $(⟨m, S_{do}(m))$ to $s$.
(6.5.15) send $(⟨P, S_{do}(P))$ to $s_1$
(6.5.16) state ← final
(6.5.17) [] state = final ∧ $∃ sbj ∈ CG (rcvd(sbj, m, S_{sbj}(m))) ∧ m = (Doc_{sbj}, n_{sbj}) ∧ ∃ Q ⊂ DG (∀ (rcvd(m, S_{sbj}(m)), dl_Q)) ∧ m = (Fpa, IMDD, n) ∧ |Q| = 2b + d + 1 ∧ (n = n_{sbj}) ∧ Full(Fpa) ∧ (Last_rcvd(Fpa) = sbj) ∧ (Ass_flow(Doc_{sbj}) = Id(Fpa)) ∧ (Id(Doc_{sbj}) = d_{id})$
(6.5.18) If (Check(Doc_{sbj}, NMReg, MReg, IMDD) = ERR):
(6.5.19) state ← recovery
(6.5.20) $m ← (err, m, S_{sbj}(m), MReg, 0)$
(6.5.21) send $(⟨m, S_{do}(m))$ to $DG$
(6.5.22) else state ← end
(6.5.23) [] state = recovery ∧ $3D ⊂ DG (∀ dl ∈ D (rcvd(m, S_{dl}(m))) ∧ (Id(Doc_{dl}) = d_{id}) ∧ m = (rec, IMDD_{dl}, MReg, Doc_{dl}, do, 0) ∧ |D| = b + 1))$
(6.5.24) $(Doc, IMDD) ← Rec-Resp-Union(\{(IMDD_{dl}, Doc_{dl})\}_{dl ∈ D}, MReg, Doc_{do})$
(6.5.25) state ← end
(6.5.26) [] state = end:
(6.5.27) $m ← (end, d_{id})$
(6.5.28) send $(m, S_{do}(m))$ to $CG ∪ DG$
(6.5.29) Replace $Doc_{do}$ with Doc in the do XML source
(6.5.30) terminate repeat

Figure 6.5: Document Originator Protocol

recovery versions and sends to $s_c$ a message (rec-merge, IMDD_{dl}, MReg, Doc_{dl}, s_c, n_{s_c}) (line 6.7.20), with $(n_{s_c} = N_{CG}[s_c])$, containing the last correct document version and corresponding IMDD structure.

- **Fpa-Checking sub-phase:** During this sub-phase $s_c$ has to send its updated $Fpa$ to delegates to be checked and signed. If no recovery has been required, $s_c$ sends a message (new-fpanr,Fpa_{s_c}) to all delegates (line 6.6.25); a message (new-fpanr,Fpa_{s_c}) is sent otherwise (line 6.6.28). $s_c$ has to send different message types if there has been a recovery or not to preserve delegates state uniqueness, that is all delegates must store the same state for the same step. Indeed if a unique message type for Fpa-checking sub-phase is required, a byzantine subject $s$ could obtain $Q$ signatures computed over this type of message and $Q$ signatures computed over a message of type rec-merge(message containing the document recovery version and
(state, first) → (initial, 0)

repeat

(6.6.03)  [] state = initial ∧ \text{recv}(\text{subj}, (m, S_{\text{subj}}(m))) ∧
\text{m = init-cg, d}_{\text{jd}}, r_{\text{subj}}, \text{doencerts}_{\text{subj}}, \text{fpmadcerts}_{\text{subj}}, \text{CG, DG, sc});

(6.6.04)  (state, MReg, N MReg, NCG[sc], do) = (wait, MDR(r_{\text{subj}}), N MDR(r_{\text{subj}}), [0...0]\text{CG, subj})

(6.6.05)  [] state = wait ∧ \text{recv}((m, S_{\text{subj}}(m))) ∧ shbj = do ∧ m = (sc, Doc, Fpa)∧
(first = 0) ∧ (\text{Id}(Doc) = d_{\text{jd}});

(6.6.06)  (state, first) → (update, 1)

(6.6.07)  [] state = wait ∧ \exists subj ∈ CG(\text{recv}((m, S_{\text{subj}}(m))) ∧ m = (Doc, n_{\text{subj}})) ∧
\exists D ∈ DG(\forall d ∈ D(\text{recv}((\text{subj}, S_{\text{subj}}))) ∧ \text{m = (Fpa, IMDD, n_{\text{subj}})}) ∧ |D| = b + 1 ∧
(NCG[subj] ≤ n_{\text{subj}}) ∧ (\text{Last-rec}(Fpa) = shbj) ∧ (\text{Last-next}(Fpa) = s_{\text{subj}}) ∧
\text{Ass-flow}(Doc) = 1d(Fpa)) ∧ (\text{Id}(Doc) = d_{\text{jd}});

(6.6.08)  NCG[subj] = (n_{\text{subj}} + 1)

(6.6.09)  If (Check(Doc, N MReg, MReg, IMDD) = ERR):

   state ← recovery

(6.6.10)  \text{m = (err, m, S_{\text{subj}}(m), MReg, NCG[s_{\text{subj}}])}

(6.6.11)  send((\text{subj}, S_{\text{subj}})) to DG

(6.6.12)  else state ← update

(6.6.13)  [] state = recovery ∧ \exists D ∈ DG(\forall d ∈ D(\text{recv}((m, S_{\text{subj}}(m))) ∧ (\text{Id}(Doc) = d_{\text{jd}})) ∧
\text{m = (rec, IMDD, MReg, Doc, sc, NCG[s_{\text{subj}}])} ∧ |D| = b + 1);

(6.6.14)  \text{m = (rec-merge, IMDD, MReg, Doc, sc, NCG[s_{\text{subj}}])} ∧ |Q| = 2b + d + 1;

(6.6.15)  state ← update-ar

(6.6.16)  [] (state = update) ∨ state = update-ar:

(6.6.17)  Doc_{\text{subj}} ← Update-Doc(Doc)

(6.6.18)  (Fpa_{\text{subj}}, s_{\text{next}}) ← Update-Fpa(Fpa)

(6.6.19)  if (state = update):

(6.6.20)  m ← (new-fpa-nr, Fpa_{\text{subj}})

(6.6.21)  send((m, S_{\text{subj}}(m))) to DG

(6.6.22)  else state ← send-fpa-nr

(6.6.23)  m ← (new-fpa-ar, Fpa_{\text{subj}})

(6.6.24)  send((m, S_{\text{subj}}(m))) to DG

(6.6.25)  state ← send-fpa-ar

(6.6.26)  [] state = send-fpa-ar ∧
\exists Q ∈ DG(\forall d ∈ Q(\text{recv}((m, S_{\text{subj}}(m))) ∧ m = (\text{signed-fpa-ar}, Fpa_{\text{subj}})) ∧ |Q| = 2b + d + 1):

(6.6.27)  m ← (fw-signed-fpa-ar, m, \{S_{\text{subj}}(m)\}_{d \in Q})

(6.6.28)  send((m, S_{\text{subj}}(m))) to DG

(6.6.29)  state ← doc-delivery

(6.6.30)  [] state = doc-delivery ∧
\exists Q ∈ DG(\forall d ∈ Q(\text{recv}((m, S_{\text{subj}}(m))) ∧ m = (\text{signed-fpa-ar}, Fpa_{\text{subj}})) ∧ |Q| = 2b + d + 1):

(6.6.31)  \text{m = (fw-signed-fpa-ar, Fpa)}

(6.6.32)  send((\text{subj}, S_{\text{subj}})) to D_{\text{subj}}

(6.6.33)  state ← doc-delivery

(6.6.34)  [] state = doc-delivery:

(6.6.35)  \text{m = (new-fpa-nr, Fpa)}

(6.6.36)  send((\text{subj}, S_{\text{subj}})) to D_{\text{subj}}

(6.6.37)  (NCG[s_{\text{subj}}], state) = (NCG[s_{\text{subj}}] + 1, wait)

(6.6.38)  [] state = wait ∧ \text{recv}((m, S_{\text{subj}}(m))) ∧ m = (end, d_{\text{jd}}) ∧ shbj = do:

(6.6.39)  terminate repeat

Figure 6.6: Subject Protocol
corresponding IMDD structure generated by delegates during the Recovery sub-phase). This could induce delegates to accept only the new Fpa or new Fpa, new document version and corresponding IMDD structure, generated by the recovery, thus producing different document versions stored by the delegates.

- **Change-Delegates-State sub-phase**: During this sub-phase, \( s \) has to notify all delegates the document recovery version and corresponding IMDD structure generated during the Recovery sub-phase, if it has been required, and the correct Fpa proposed during the Fpa-checking sub-phase. Two message types are provided by our protocols: a message \((\text{fw-signed-fpa-nr}, m, \{S_{dl}(m)\}_{dl \in Q})\) (line 6.6.31), with \( m \) message of type new-fpa-nr, containing the new Fpa and \(|Q| = Q\) signatures computed on \( m \); and a message \((\text{fw-signed-fpa-ar}, \overline{m}, \{S_{dl}(\overline{m})\}_{dl \in \overline{Q}}, \hat{m}, \{S_{dl}(\hat{m})\}_{dl \in \hat{Q}}\) (line 6.6.37), with \( \overline{m} \) message of type new-fpa-ar, containing the new Fpa and \( \hat{m} \) message of type rec-merge containing the document recovery version and associated IMDD structure and \(|\overline{Q}| = Q\) signatures computed on \( \overline{m} \) and other \(|\hat{Q}| = Q\) signatures computed on \( \hat{m} \). As we will state formally in what follows a subject cannot generate a valid message of type \( \text{fw-signed-fpa-nr} \) and another valid message of type \( \text{fw-signed-fpa-ar} \) in the same step, because the protocols prevent this subject from collecting \( Q \) signatures for a message of type \( \text{new-fpa-nr} \) and \( Q \) signatures for a message of type \( \text{new-fpa-ar} \). Indeed a delegate does not accept, in the same step, messages with the same priority and different content.

- **Down-Delegate-agreement sub-phase**: During this sub-phase a down delegate \( dl_{\text{down}} \), that wakes up at time \( t \) in step \( x \), waits for a time equal to \( MTIME \) (line 6.8.04), before sending a message (agreement) to all delegates, to allow messages sent before \( t \), and not received by \( dl_{\text{down}} \), to be received by all operative delegates. This method assures that the lost messages will be inserted in the responses sent by operative delegates to \( dl_{\text{down}} \). These responses are used by \( dl_{\text{down}} \) to locally make stable \( \text{State}^y \), with \( y \geq x \). This is due to the delay between the time in which the agreement request is sent and the time in which a down delegate receives the required responses and process them. Moreover a down delegate after processing received responses, processes also all the messages received after \( t \). Each delegate at step \( z \), during a Change-Delegates-State sub-phase, stores in a local repository the new state, \( \text{State}^{z+1} \). If a delegate goes down at step \( z \), before reaching or completing the Change-Delegates-State sub-phase, it has in its local repository the last state it made stable, \( \text{State}^z \), that will be used during the Down-Delegate-agreement sub-phase to evaluate received
Chapter 6: Cooperative updates in colluding byzantine and failure prone distributed systems

(6.7.01) (state, requests, IMDDst, history) — (initial, 0, 0, 0)

(6.7.02) repeat

(6.7.03) [] state = initial \land rcvd(sbj, ⟨m, Ssbj(m)⟩) \land
m = (init-dg, did, Docact, Fparw, CG, DG) \land (dlc \in DG):

(6.7.04) (state, TP, NTP, do) — (norec, CG \cup DG \cup \{do\}, [0 \ldots 0]TP, sbj)

(6.7.05) LSRR — (1{do(Fpa(st))}, [0 \ldots 1{do(Fpa(st))}, 0]MDR, Docact)

(6.7.06) [] (state = norec \land state = rec) \land \exists dl \in DG rcvd(m, Sgl(m)) \land m = (agreement):

(6.7.07) Let agreements be the set of not still processed messages of type agreement present in Queue

(6.7.08) \(\hat{m}\) — (agreement-resp, history, hpm, agreements, dl, NTP[dl])

(6.7.09) send((\hat{m}, Sdl(\hat{m}))) to dl

(6.7.10) NTP[dl] — NTP[dl] + 1

(6.7.11) [] state = norec \land \exists sbj, sbj \in CG rcvd(⟨m, Ssbj(m)⟩) \land
m = (err, \(\hat{m}\), Sgl(\(\hat{m}\)), MReg, n_sbj) \land (Last_next(Fparw) = sbj) \land (NTP[sbj] = n_sbj) \land (requests = 0):

(6.7.12) (state, hpm, IMDDrec) — (rec, (m, Ssbj(m)), IMDDst)

(6.7.13) (Docact, IMMDdl, IMDDc) — Recovery(Doc, MReg, IMDDrec, Docact, LSRR)

(6.7.14) \(\bar{m}\) — (rec-merge, IMMDdl, MReg, Docact, sbj, n_sbj)

(6.7.15) send((\(\bar{m}\), Sdl(\(\bar{m}\)))) to sbj

(6.7.16) [] (state = rec) \land \exists sbj \in CG, D \subset DG rcvd(⟨m, Ssbj(m)⟩) \land
m = (fw-rec, (\(\bar{m}\), Sgl(\(\bar{m}\))) , D) \land \(\bar{m}\) = (rec, IMMDtl, MReg, Docact, sbj, n_sbj) \land
\(\bar{m}\) = (Last_next(Fparw) = sbj) \land (NTP[sbj] = n_sbj) \land (requests = 0):

(6.7.17) (requests, state, Rec-Doc, hpm) — (1, rec, Docact, (m, Ssbj(m)))

(6.7.18) (Docact, IMDDdl, IMDDc) — Rec-Resp-Merge(⟨IMMDdl, Docact⟩) ∈ D, MReg, RecDoc

(6.7.19) \(\overline{m}\) — (rec-merge, IMMDdl, MReg, Docact, sbj, n_sbj)

(6.7.20) send((\(\overline{m}\), Sdl(\(\overline{m}\)))) to sbj

(6.7.21) [] state = norec \land \exists sbj \in CG rcvd(⟨m, Ssbj(m)⟩) \land m = (new-fpa-ar, Fpa) \land
\(\text{Last_next}(Fparw) = sbj) \land (\text{Recvrs}(Fpa) = \text{Recvrs}(Fpa) + 1)

(requests = 0):

(6.7.22) (requests, hpm, state) — (1, (m, Ssbj(m)), norec)

(6.7.23) If (Check-Fpa(Fpa)) \neq ERR:

(6.7.24) send((\(m\), Sgl(\(m\)))) to Fpa

(6.7.25) (state, requests, IMDDst, history) — (initial, 0, 0, 0)

(6.7.26) [] state = rec \land \exists sbj \in CG rcvd(⟨m, Ssbj(m)⟩) \land m = (new-fpa-ar, Fpa) \land
\(\text{Last_next}(Fparw) = sbj) \land (\text{Recvrs}(Fpa) = \text{Recvrs}(Fpa) + 1)

(requests = 1):

(6.7.27) (requests, hpm, state) — (2, (m, Ssbj(m)), rec)

(6.7.28) If (Check-Fpa(Fpa)) \neq ERR:

(6.7.29) send((\(m\), Sgl(\(m\)))) to sbj

(6.7.30) (state, requests, IMDDst, history) — (initial, 0, 0, 0)

(6.7.31) [] (state = norec \land state = rec) \land \exists sbj \in CG, Q \subset DG rcvd(⟨m, Ssbj(m)⟩) \land
|Q| = (2b + d + 1) \land m = (fw-signed-fpa-ar, \(\overline{m}\), Sgl(\(m\))) \land \(\overline{m}\) = (signed-fpa-ar, Fpa) \land (\text{Recvrs}(Fpa) = \text{Recvrs}(Fpa) + 1) \lor
\exists sbj \in CG, Q1, Q2 \subset DG rcvd(⟨m, Ssbj(m)⟩) \land |Q1| = |Q2| = (2b + d + 1) \land m = (fw-signed-fpa-ar, \(\overline{m}\), Sgl(\(m\))) \land \(\overline{m}\) = (signed-fpa-ar, Fpa) \land \(\overline{m}\) = (rec-merge, IMMD, MReg, Doc, sbj, n_sbj) \land
(\text{Recvrs}(Fpa) = \text{Recvrs}(Fpa) + 1) \land (NTP[sbj] = n_sbj) \land (requests \leq 2):

(6.7.32) (requests, state, hpm, history) — (0, norec, (m, Ssbj(m)), history \cup \{(m, Ssbj(m))\})

(6.7.33) send((m, Ssbj(m))) to DG

(6.7.34) If (m = (fw-signed-fpa-ar, \(\overline{m}\), Sgl(\(m\)))):

(6.7.35) (Fparw, NTP[Last\_recv(Fpa)]) — (Fpa, NTP[Last\_recv(Fpa)] + 1)

(6.7.36) else

(6.7.37) (Fparw, Docact, IMMDtl, LSRR, NTP) — Stable(Fpa, Doc, IMMD, NTP, MReg)

(6.7.38) (m = (fw-signed-fpa-ar, \(\overline{m}\), Sgl(\(m\)))) to Last\_next(Fparw)

(6.7.39) send((\(\overline{m}\), Sdl(\(\overline{m}\)))) to Last\_next(Fparw)

(6.7.40) [] (state = norec \land state = rec) \land rcvd(⟨m, Ssgl(m)⟩) \land m = (end, d, did):

(6.7.41) terminate repeat

Figure 6.7: Delegate Protocol
messages and choose if processing or discarding them. Operative delegates send a response message \( m = (\text{agreement-resp}, \text{history}, \text{hpm}, \text{agreements}, \text{dl}_{down}, N_{TP}[\text{dl}_{down}]) \), where: \( \text{history} \) contains all the messages of type \( \text{fw-signed-fpa-nr} \) or \( \text{fw-signed-fpa-ar} \) received by an operative delegate from the beginning of the process, \( \text{hpm} \) contains the last accepted and processed higher priority message, \( \text{agreements} \) contains all the received and not yet processed messages of type \( \text{agreement} \). \( m.\text{dl}_{down} \) and \( N_{TP}[\text{dl}_{down}] \) are information used to avoid a reply attack. \( \text{dl}_{down} \) accepts the first \((b + 1)\) received responses coming exactly from \((b + 1)\) distinct delegates (line 6.8.08). After that, \( \text{dl}_{down} \), processing only once all valid messages in the \((b + 1)\) received \( \text{history} \) components, makes stable the most recent state made stable by at least one delegate, among the \((b + 1)\) delegates whose response has been accepted by \( \text{dl}_{down} \). At this point \( \text{dl}_{down} \) processes messages of type \( \text{fw-signed-fpa-nr} \) or \( \text{fw-signed-fpa-ar} \) until structure \( \text{Queue} \) contains no more messages of these types, making stable in sequence all states made stable by the operative delegates after the delivery of their responses. Then, \( \text{dl}_{down} \) processes the message in \( \text{Queue} \) chosen according to the received messages schedul-

Figure 6.8: Down Delegate Protocol
### Table 6.18: Functions (part 1)

<table>
<thead>
<tr>
<th>Function</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full</td>
<td>it returns <em>true</em>, if $Fpa$ has been completed (there is no free element), <em>false</em> otherwise</td>
</tr>
<tr>
<td>last_recv</td>
<td>it finds the public key associated with the subject that has proposed $Fpa$ as current version (last subject registered in $Fpa$).</td>
</tr>
<tr>
<td>Ass_flow</td>
<td>it returns the identifier of $Fpa$ associated with $Doc$</td>
</tr>
<tr>
<td>Id</td>
<td>it returns the identifier associated with $Doc$ or $Fpa$</td>
</tr>
<tr>
<td>Check</td>
<td>it checks the document ($Doc$) integrity according to IMDD structure and wrt the set of non-modifiable ($NMReg$) and modifiable ($MReg$) regions</td>
</tr>
<tr>
<td>Rec-Resp-Union</td>
<td>it inserts in Doc the last correct version of regions in $MReg$, using region versions and invalid declarations produced by (b+1) delegates</td>
</tr>
<tr>
<td>Last_next</td>
<td>it finds the public key associated with the subject chosen by the last receiver to be the next receiver</td>
</tr>
<tr>
<td>Update-Doc</td>
<td>it allows a subject to modify $Doc$ according to rights it possesses</td>
</tr>
<tr>
<td>Update-Fpa</td>
<td>it allows a subject to modify $Fpa$ according to rights it possesses, insert its information in $Fpa$, and choose the next receiver ($s_{next}$)</td>
</tr>
<tr>
<td>MDR / NMDR</td>
<td>they extract respectively modifiable and non-modifiable regions from a document or a structure containing regions and keys</td>
</tr>
<tr>
<td>Recovery</td>
<td>it collects last correct modifiable regions content asking to the subjects, that has modified these regions, their last generated region versions</td>
</tr>
<tr>
<td>Rcvrs</td>
<td>it returns the number of receivers that has already received $Fpa$</td>
</tr>
<tr>
<td>Check-Fpa</td>
<td>it checks the integrity of $Fpa$</td>
</tr>
<tr>
<td>Stable</td>
<td>it replaces previous state at step $x$ ($State^x$) with the subsequent state ($State^{x+1}$)</td>
</tr>
</tbody>
</table>

The functions, here applied with the variant of no taking into account, in the choice of messages, variables *state* and *requests* wrt the policy used in the Delegate Protocol. This is done for example to allow a down delegate to process a message of type *new-fpa-ar* (lines 6.7.27-32), indicated in the component *hpm* of some responses, also having its variables *state* and *requests* set to *norec* and 0 respectively, according to the previous processing of a message of type *fw-signed-fpa-nr* or *fw-signed-fpa-ar* (lines 6.7.33-41) and allow the update process to go on (for example generation of the $Q^b$ distinct signature on message $m = (new-fpa-ar,Fpa)$ required by $s_c$ to start the Change-Delegates-State sub-phase).
Table 6.19: Functions (part 2)

<table>
<thead>
<tr>
<th>Function</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock</td>
<td>it return the value of the internal clock</td>
</tr>
<tr>
<td>Extract-msg</td>
<td>it inserts in structure Queue messages of type fw-signed-fpa-nr or fw-signed-fpa-ar contained in history structures received during an agreement phase</td>
</tr>
<tr>
<td>Received-msg</td>
<td>it adds to Queue messages received after the last execution of this function</td>
</tr>
<tr>
<td>Higher-priority</td>
<td>it finds in structure Queue the message with higher priority that satisfies a delegate condition statement, regardless the values for state and requests</td>
</tr>
<tr>
<td>Valid</td>
<td>it returns true if the analyzed message satisfies at least one delegate condition statement, regardless the values for state and requests, false otherwise</td>
</tr>
<tr>
<td>Priority</td>
<td>it returns the priority associated with the message according to the priorities specified in Tables 6.16 and 6.17</td>
</tr>
<tr>
<td>Type</td>
<td>it returns the type of the message according to types specified in Tables 6.16 and 6.17</td>
</tr>
<tr>
<td>Execute</td>
<td>it executes the statements associated with the delegate condition statement satisfied by the analyzed message and its signature, regardless the values for state and requests; if it receives an empty message it sets: state ← norec, requests ← 0</td>
</tr>
</tbody>
</table>

After that $d_{\text{down}}$ has made stable the state common to all operative delegates and set its variables $\text{state}$ and $\text{requests}$ to the values assigned to them by the execution of the last processed message. The Down Delegate protocol ends and $d_{\text{down}}$ becomes operative.

Note that if the following assumptions hold: 1) $d_{\text{down}}$ goes down in step $x$; and 2) it wakes up in the same step; and furthermore 3) let $m$ be the higher priority message $d_{\text{down}}$ has processed before going down; then $d_{\text{down}}$ does not process, in step $x$, messages of type different by agreement with: a) priority lower than that associated with $m$, or b) priority equal to that associated with $m$, but with different content. This is done to avoid the situation in which a delegate signs a message of a particular type (for example new-fpa-ar) and then, when it wakes up, it signs a message with the same priority (for example new-fpa-nr) giving the possibility to a byzantine subject to induce disjoint subsets of delegates to make stable different states.
6.5.3.3 Final phase

The do receives, by the last receiver (sj) in Fpa, a message m containing the document (Doc_sbj) and another message by the delegates containing Fpa (line 6.5.17). It checks the document integrity and, if an error occurs, it sends to all delegates an error message (err, m, S sbj(m), M Reg, 0) (line 6.5.21). Each delegate dl ∈ DG generates a document recovery version contacting subjects in CG and then sends a message (rec, IMDD dl, M Reg, Doc dl, do, 0), containing this version, to do. do accepts the first (b + 1) messages from the delegates and then composes them to obtain the last correct document version. In this way do is sure that the content of all modifiable regions is the last correct one retrieved by at least one operative delegate or a more recent correct version. The original document is thus replaced by this new document (line 6.5.29). At this point do sends a message (end, d id) to all delegates and subjects, to end the cooperative update process concerning document with identifier equal to d id (line 6.5.28).

6.5.4 Formal results

We are now ready to present some formal results regarding our protocols.

Lemma 7 (Message Execution Uniqueness) Let dl ∈ OP be an operative delegate. Delegate and Down Delegate protocols assure the following properties:

a) If dl processes a message with priority p in a step x, with x ≥ 0, then in the same step it will not process messages with priority less than or equal to p and type different by agreement;

b) If the following conditions hold: 1) dl process, in a step x, with x ≥ 0, a message m with content c and priority p, such that m is the higher priority message processed in step x by dl; and 2) after that it goes down; and 3) all operative delegates are in the same step x, until dl becomes once again operative; then dl will not process, in step x, messages of type different by agreement with: a) priority less than p or b) priority equal to p and content different by c.

Theorem 8 (Delegate State Uniqueness) Let dl, dlg ∈ OP be operative delegates. If dl makes stable State x dl and dlg makes stable State x dlg, then State x dl = State x dlg, with x ≥ 0.

Proposition 9 (Down delegates caused delay freeness) The maximum number of down delegates that do not delay the system is d.

---

5More details about the recovery functionality and the composition of recovery versions are presented in Section ??.
Proposition 10 (System survivability) The system survives, that is it is definitely able to perform each required sub-phase, according to the protocols, in presence of a number of down delegates less than or equal to \((b+2d)\).

6.5.5 Recovery

The Recovery sub-phase has the aim of retrieving the last correct version of the document content associated with the accessible regions of \(s_c\), contacting subjects in \(CG\) that have received till that point at least a document package and that have executed at least a modification operation on at least one accessible region of \(s_c\). Correctness of a region is determined according to the set of valid modification declarations inserted in \(Fpa\), thus we have first to introduce some important concepts, such as modification declarations validity and region state correctness, used in the recovery process. After that, we present the two most important and innovative algorithms, among all the algorithms implied by our approach. In particular these algorithms are: the recovery algorithm (Recovery) shown in Figures 6.9 and 6.10, used by a delegate to generate a document recovery version and the corresponding IMDD structure (invoked in line 6.7.13), and the recovery responses merge algorithm (Rec-Resp-Merge) shown in Figure 6.11, used by a delegate to compose the received \((b+1)\) document recovery versions and associated IMDD structures (invoked in line 6.7.18).

6.5.5.1 Preliminary definitions

First, we have to define validity of modification declarations, that is conditions that make them valid wrt a document version. We start by formally defining validity of delete elemt modification declarations wrt a document version.

**Definition 18 (delete elemt modification declaration validity wrt a document version).** Let \(dv\) be a document version of the document to be updated, and IMDD the associated structure for invalid declarations. Let \(ded\) be a delete elemt modification declaration, containing a doc-root id, inserted by a subject \(s\) in its receiver specification of \(Fpa\). \(ded\), such that \(\neg(ded \in IMDD)\), is valid wrt \(dv\) if one of the following conditions holds: 1) a certificate \(c\) containing subject \(s\), object doc-root_id, privilege delete elemt and correct wrt its associated signature exists; 2) a certificate \(c'\) with doc-root_id \(\in c'.obj.reg[r].set_docae\), where \(r \in c'.obj.reg\{r_id\}\), correct wrt its associated signature exists and another delete elemt modification declaration valid wrt \(dv\) according to \(c'\) exists.

Previous definition states that a declaration of deletion of a document sub-tree is valid if a certificate corresponding to this declaration exists or a certificate corresponding to a deletion dec-
laration of a document sub-tree that includes the previous one exists. In what follows, we formally define validity of a delete\_attr modification declaration wrt a document version.

**Definition 19 (delete\_attr modification declaration validity wrt a document version).** Let \( dv \) be a document version of the document to be updated, and \( \text{IMDD} \) the associated structure for invalid declarations. Let \( \text{dad} \) be a delete\_attr modification declaration associated with a modifiable region \( r \) in \( dv \), inserted by a subject \( s \) in its receiver specification at position \( p \) in \( \text{Fpa} \) and such that \( \neg (\text{dad} \in \text{IMDD}) \). \( \text{dad} \) is valid wrt \( dv \) if one of the following conditions holds: 1) a certificate \( c \) containing subject \( s \), region \( r \), privilege delete\_attr and correct wrt its associated signature exists and also the union of all atomic elements declared in this modification declaration and in those of delete\_attr for region \( r \), inserted in positions of \( \text{Fpa} \) less than \( p \) and not present in \( \text{IMDD} \), are contained in the set of atomic elements of \( r \) that are attributes; 2) a delete\_attr modification declaration, valid wrt \( dv \), inserted in a positions of \( \text{Fpa} \) greater than \( p \) exists and \( \text{dad} \) is not present in \( \text{IMDD} \).

This definition states that a delete\_attr modification declaration is valid if the subject that has inserted this declaration in \( \text{Fpa} \) has also inserted in document control information of the currently analyzed document version a corresponding certificate and the elements declared as deleted by this declaration and all previous inserted delete\_attr modification declarations, not present in the set of invalid declarations, belong to the set of attributes of the modified region, since only these elements can be deleted according to this privilege. A delete\_attr declaration is also valid if it is not evaluated as invalid till that point and another valid subsequent delete\_attr declaration exists in \( \text{Fpa} \). Also an invalid declaration, because corresponding to a deletion of some attributes without the possession of a proper certificate, but not still evaluated as invalid by the protocol, and thus not inserted in \( \text{IMDD} \), will be considered valid if a subsequent subject \( s \) will correctly exercise the delete\_attr privilege on the same region. Thus, in general for the delete\_attr privilege, but also for the update\_attr one, validity of a declaration makes valid previous declarations that are not yet evaluated as invalid. For the delete\_elem privilege is the same, only in the case in which a new valid declaration for the deletion of a sub-tree, and corresponding certificate, encapsulates the sub-tree specified in a previous invalid, but not yet evaluated as invalid, declaration. In the following we formally define validity of an update\_attr modification declaration declaration wrt a document version.

**Definition 20 (update\_attr modification declaration validity wrt a document version).** Let \( dv \) be a document version of the document to be updated, and \( \text{IMDD} \) the associated structure for invalid declarations. Let \( \text{uad} \) be an update\_attr modification declaration associated with a modifiable
region $r$ in $dv$, inserted by a subject $s$ in its receiver specification of $Fpa$, and such that $\neg (dad \in IMDD)$. $uad$ is valid wrt $dv$: 1) if a certificate $c$ containing subject $s$, region $r$, privilege update attr and correct wrt its associated signature exists and the content of each atomic element in $r$, not declared as deleted by a valid delete attr/delete elemt modification declaration, corresponds to the associated component $h_{docae}$, and a signature computed on components $h_{docae}$ by $s$ exists and it is correct; or 2) if an update attr modification declaration, valid wrt $dv$, inserted in a subsequent receiver specification in $Fpa$ exists.

This definition states that an update attr declaration is valid in presence of a proper certificate and when the content of $h_{docae}$ components associated with elements of the modified region are correct wrt the signature computed by the subject who generated the declaration. Moreover content of atomic elements not declared as deleted must be correct wrt the corresponding $h_{docae}$ component. As above-mentioned an update attr declaration is also valid if another subsequent update attr declaration is valid. In the following we formally define validity of a set of modification declarations wrt a set of document versions.

**Definition 21 (validity of a set of modification declarations wrt a set of document versions).** Let $MD$ be a set of modification declarations in $Fpa$ and let $DV$ be a set of document versions. $MD$ is valid wrt $DV$ if $\forall md \in MD \exists dv \in DV$ (md is valid wrt dv).

This definition completes the notion of validity of a modification declaration wrt to a document version, generalizing it in terms of validity of a set of declarations wrt to a set of document versions. Now we are ready to define the concept of correctness associated with a region state wrt a set of valid modification declarations.

**Definition 22 (region state correctness wrt a set of valid modification declarations).** Let $r$ be a modifiable region. Let $MD$ be the set of modification declarations in $Fpa$, associated with $r$, valid wrt a set $DV$ of document versions. Let $r$’s region state be the $r$’s region content together with the $r$’s region control information. $r$’s region state is correct wrt valid modification declarations in $MD$ if all the following conditions hold: 1) $r$’s region state is that contained in the document version in $DV$ that makes valid the most recent update attr modification declaration in $MD$ or it is that contained in the original document, in absence of valid update attr modification declarations in $MD$; 2) all atomic elements belonging to $r$, declared as deleted by delete elemt modification declarations in $MD$ have a null value; 3) all atomic elements belonging to $r$, declared as deleted
Chapter 6: Cooperative updates in colluding byzantine and failure prone distributed systems

by delete\_attr modification declarations in MD have a null value; 4) r’s region state contains all certificates that make valid all modification declarations in MD.

This definition states that a region state, correct wrt a set of valid modification declarations, consists of all certificates that makes valid those declarations, that is region state is generated according to correct possessed privileges. Moreover it consists of a content, corresponding to associated h\_docae components correct wrt the signature computed over them by the subject corresponding to an update\_attr certificate, that assures that it is generated by an authorized subject. Furthermore this content must also be that generated by the last authorized subject that has correctly modified the region and with all the elements declared as deleted by the considered set of modification declarations set to null.

6.5.5.2 Recovery algorithm

In what follows we give a brief description of the recovery algorithm presented in Figures 6.9 and 6.10. Initially Algorithm 2 replaces all non-modifiable information in the document to be recovered (\(Doc_0\)), with those in the stable version of the document (\(Doc_{st}\)). At point 2 function Rec\_MDD\_Collection collects all modification declarations, not yet inserted in the stable version of IMDD and associated with at least a region in MReg, in structure MDD. Each delete\_attr/delet\_attr declaration, among the previous selected ones, is collected if it is present in a position of Fpa greater than the position stored in LSRR corresponding to the region to which the modification declaration is associated with. Each delete\_attr modification declaration (\(dad\)) in MDD associated with a region \(r\) contains: 1) the \(r\)’s identifier, 2) the cumulative set of elements declared as deleted, that is the union of the sets of elements declared as deleted by all the delete\_attr declarations, for \(r\), in MDD that precede \(dad\) in Fpa and the set of elements specified in \(dad\), and 3) the public key of the subject that has generated \(dad\). In MDD are also inserted, for each region in MReg, the most recent valid update\_attr/delete\_attr declarations determined during the last recovery sub-phase.

The algorithm considers the position in Fpa corresponding to the subject (\(sb \cdot j(Doc)\)) that has generated the corrupted document to be recovered as the initial recovery position. Component set-del-docae will contain all the elements declared as deleted in valid delete\_elem/delete\_attr declarations. Algorithm 2 analyzes declarations in MDD until the set is empty and when required it contacts a subject to obtain a document version against which evaluating the declarations in MDD. Algorithm 2 starts by the document version to be recovered and then, if required, document versions
Algorithm 2: The recovery protocol (Recovery)

INPUT: Doc: document version to be recovered

MReg: set of modifiable regions to be recovered

MDD: structure containing all the Modification Document Declarations

IMDD: Invalid Modification Document Declaration structure

Doc_{st}: Stable Document (document containing the last recovered contents)

LSRR: LastSaved–RegionRecovery structure

OUTPUT: (Doc, IMDD): the recovered document and the set of invalid modification document declarations

METHOD:

1. *replacement of the non-modifiable information in the corrupted received package*

Replace all non-modifiable information (also those in the modifiable regions) within Doc with those stored in Doc_{st}

2. *collection of modification declarations and initialization of algorithm parameters*

MDD ← Rec-MDD-Collection(MDD, IMDD, MReg, LSRR)

set-del-docae ← ∅ * set of elements declared as deleted *

Sbj ← {sbj(Doc)} * set of subjects from which has been already received a document version, initialized to the subject that generated document version Doc *

Rec-Position ← document-position(Doc) * position of sbj(Doc) in the Fpa *

D ← Doc_{st}

3. *declarations validity check and last correct document contents search*

While (MDD ≠ ∅):


For each c ∈ Doc://delete_elmt_cert such that (c.obj.reg{r_id} ∩ MReg ≠ ∅):

If (D_{KU}(c.signature) ≠ H(c.obj.reg{r_id}) ∨ c.doc.id ≠ doc.id):

Doc://delete_elmt_cert ← Doc://delete_elmt_cert \ {c}

Else If (∃ d ∈ MDD: Del-Elmt: (c.obj.roo_id = d.id) ∧ (c.sbj = sbj(Doc))):

Doc://delete_elmt_cert ← Doc://delete_elmt_cert \ {c}

For each c ∈ delete_elmt_cert such that (c.obj.reg{r_id} ∩ MReg ≠ ∅):

If (∃ e ∈ delete_elmt_cert \ {c} ≠ e ∧ ∀ r ∈ c.obj.reg{r_id}):

c.obj.reg[r].set-docae ⊆ c.obj.reg[r].set-docae:

Doc://delete_elmt_cert ← Doc://delete_elmt_cert \ {c}

4. *check that all delete element declarations have associated a certificate containing the delete_elmt privilege and saving in set-del-docae component the set of declared deleted elements*

For each del-decl ∈ MDD: Del-Elmt:

If ((∃ c ∈ Doc://delete_elmt_cert: c.obj.roo_id = del-decl.doc.roo_id) ∨

(∃ c ∈ Doc://delete_elmt_cert: c.obj.roo_id = del-decl.doc.roo_id) ∧

∃ r ∈ c.obj.reg{r_id}: del-decl.doc.roo_id ∈ c.obj.reg[r].set-docae)

MDD: Del-Elmt ← MDD: Del-Elmt \ {del-decl}

For each r ∈ (c.obj.reg{r_id}):

set-del-docae ← set-del-docae ∪ c.obj.reg[r].set-docae

5. *delete_attr declarations evaluation*

For each r ∈ MDD: Del-Attr{r_id}:

Let Del-Attr{r_id} be the declaration in MDD: Del-Attr for the region r with the highest position

If (position(Del-Attr{r_id}) ≤ position(LSRR{r})): Doc.FDUR{r}.delete_attr_cert ← Doc_{st}.FDUR{r}.delete_attr_cert

set-del-docae ← set-del-docae ∪ Del-Attr{r_id}.set-docae

MDD: Del-Attr ← MDD: Del-Attr \ {Del-Attr{r_id}}

else Let Attr-docae(r) ⊆ Doc_{st}.FDUR{r}.DAE.LIST{docae.id} be the set of document atomic elements belonging to the fully deletable and updatable region r that are attributes

If ((∀ c ∈ D.FDUR{r}.delete_attr_cert: (c.obj = r) ∧ (c.sbj = sbj) ∨ (c.doc.id = doc.id) ∧

(c.priv = delete_attr) ∧ (D_{KU}(c.signature) = H(c.signature))): Doc.FDUR{r}.delete_attr_cert ← \{c\}

set-del-docae ← set-del-docae ∪ Del-Attr{r_id}.set-docae

For each del_attr{r_id} ∈ MDD: Del-Attr containing region r:

MDD: Del-Attr ← MDD: Del-Attr \ {del_attr{r_id}}

Figure 6.9: The recovery algorithm (part 1)
6. *update attr declarations evaluation*

   **For each** $r \in \text{MDD.\text{Up-Attr}}\{\text{id}\}$:
   
   Let $\text{Up\_attr\_decl}$ be the declaration in $\text{MDD}\_\text{Up-Attr}$ for the region $r$ with the highest position.

   If $(\text{position}(\text{Up\_attr\_decl}) \leq \text{position}(\text{LSRR}(r)))$:
   
   $\text{Doc.FDUR}[r].\text{update\_cert} \leftarrow \text{Doc}_{\text{attr}}.\text{FDUR}[r].\text{update\_cert}$
   
   $\text{MDD.\text{Up-Attr}} \leftarrow \text{MDD.\text{Up-Attr}} \setminus \{\text{Up\_attr\_decl}\}$
   
   Insertion in Doc of the modifiable content of region $r$ contained in Doc$_{\text{attr}}$

   Else $\text{chk} \leftarrow \text{Correct}$

   **For each** $\text{docae} \in (\text{P.FDUR}[r].\text{DAE\_LIST}\{\text{docae}\_\text{id}\}) \setminus \text{set\_del\_docae}$:
   
   Compute $h_{\text{docae}} \leftarrow H(\text{D.FDUR}[r].\text{DAE\_LIST}[\text{docae}].\text{data})$
   
   If $(h_{\text{docae}} \neq D.\text{FDUR}[r].\text{DAE\_LIST}[\text{docae}].h_{\text{docae}})$: $\text{chk} \leftarrow \text{ERR}$
   
   Break

   **If** $(\text{DKU}_{\text{up\_attr\_decl}.\text{pub\_key}}\{\text{sig}\_\text{f\_d\_docae}\}) = H(\bigcup_{\text{decl} \in \text{MDD}} (\text{Up\_attr\_decl}\_\text{fp\_id}\_\text{signature} \times \text{Up\_attr\_decl}\_\text{orig})) \land (\text{chk} = \text{Correct}) \land$

   $(\exists d \in \text{D.FDUR}[r].\text{update\_cert}: (\text{Up\_attr\_decl}\_\text{pub\_key} = \text{c.sbj}\_\text{pub}\_\text{key}) \land$

   $(\text{c.obj} = \text{t}) \land (\text{c.doc\_id} = \text{doc\_id}) \land (\text{c.priv} = \text{update\_attr}) \land$

   $(\text{DKU}_{\text{c.sbj\_signature}}(c.\text{signature}) = H(c.\text{signature})))$:
   
   $\text{Doc.FDUR}[r].\text{update\_cert} \leftarrow \{\text{c}\}$
   
   Insertion in Doc of the modifiable content of region $r$ contained in Doc$_{\text{attr}}$

   **For each** $\text{up\_attr\_decl} \in \text{MDD}\_\text{Up-Attr}$ containing region $r$:
   
   $\text{MDD.\text{Up-Attr}} \leftarrow \text{MDD.\text{Up-Attr}} \setminus \{\text{up\_attr\_decl}\}$

7. *evaluation of invalid declarations and search of the next subject to be contact*

   **For each** $\text{decl} \in \text{MDD}$:
   
   If $(\text{position}(\text{decl}) = \text{Rec-Position})$:
   
   $\text{IMDD} \leftarrow \text{IMDD} \cup \{\text{decl}\}$
   
   $\text{MDD} \leftarrow \text{MDD} \setminus \{\text{decl}\}$
   
   Let $\text{decl\_set}$ be the set of declarations in $\text{MDD}$ with the same highest position (declarations of the same subject in a position of the $\text{Fp}$)

   While $(\text{decl\_set} \neq \text{null})$:

   **If** $(\exists \text{d} \in \text{decl\_set}: (\text{d} \in \text{MDD}\_\text{Up-Attr} \lor \text{d} \in \text{MDD}\_\text{Del-Attr}) \land$

   $(\text{position}(\text{d}) > \text{position}(\text{LSRR}(\text{d}\_\text{obj}\_\text{id}) \lor \text{d} \in \text{MDD}\_\text{Del-Elmt}))$:
   
   **If** $(\text{d.pub\_key} \in \text{Sbj})$:
   
   $\text{IMDD} \leftarrow \text{IMDD} \cup \{\text{doc\_set}\}$
   
   $\text{MDD} \leftarrow \text{MDD} \setminus \{\text{decl\_set}\}$
   
   $\text{decl\_set} \leftarrow \text{prev}(\text{decl\_set}, \text{MDD})$

   Else $\text{send}(\text{doc\_id}, \text{doc\_version}, \text{doc\_orig}, \text{pack\_req})$ to $\text{d.pub\_key}$

   **If** $(\text{recv}(\text{d.pub\_key}, \text{D}) \lor \text{end}(\text{Timeout}))$:
   
   **If** $(\text{end}(\text{Timeout}))$:
   
   $\text{IMDD} \leftarrow \text{IMDD} \cup \{\text{decl\_set}\}$
   
   $\text{MDD} \leftarrow \text{MDD} \setminus \{\text{decl\_set}\}$
   
   $\text{decl\_set} \leftarrow \text{prev}(\text{decl\_set}, \text{MDD})$

   Else $\text{Rec-Position} \leftarrow $ position$(\text{d})$

   $\text{decl\_set} \leftarrow \text{null}$

   Else $\text{For each} \text{de} \in \text{decl\_set}$:

   **If** $(\text{de} \in \text{MDD}\_\text{Up-Attr})$:

   $\text{Doc.FDUR}[r].\text{update\_cert} \leftarrow \text{Doc}_{\text{attr}}.\text{FDUR}[r].\text{update\_cert}$
   
   $\text{MDD.\text{Up-Attr}} \leftarrow \text{MDD.\text{Up-Attr}} \setminus \{\text{de}\}$

   Insertion in Doc of the modifiable content of region $r$ contained in Doc$_{\text{attr}}$

   Else $\text{Doc.FDUR}[r].\text{delete\_attr\_cert} \leftarrow \text{Doc}_{\text{attr}}.\text{FDUR}[r].\text{delete\_attr\_cert}$

   set-del-docae $\leftarrow \text{set-del-docae} \cup \text{de.set-docae}$

   $\text{MDD}\_\text{Del-Attr} \leftarrow \text{MDD}\_\text{Del-Attr} \setminus \{\text{de}\}$

   $\text{decl\_set} \leftarrow \text{prev}(\text{decl\_set}, \text{MDD})$

8. *deletion of document atomic element contents declared as deleted*

   **For each** $\text{docae} \in \text{set-del-docae}$:

   Let $\text{reg}$ be the region such that: $\text{docae} \in \text{Doc.FDUR}[\text{reg}].\text{DAE\_LIST}\{\text{docae}\}$

   $\text{Doc.FDUR}[\text{reg}].\text{DAE\_LIST}[\text{docae}].\text{data} \leftarrow \text{null}$

9. *return of the algorithm result*

   $\text{return}(\text{Doc}, \text{IMDD})$

---

Figure 6.10: The recovery algorithm (part 2)
are obtained in reverse order wrt the order associated with receiver specifications in $F_{pa}$.

First, it analyzes certificates associated with delete-elem privilege. It removes all certificates that are not correct or that are not associated with a delete-elem declaration or that are contained in another certificate. Then, delete-elem declarations are evaluated valid if a corresponding certificate exists as specified in Definition 18, and elements declared as deleted are saved in set-del-docae component.

Point 5 analyzes each region for which there exists a delete_attr declaration in $MDD$. It determines the most recent delete_attr declaration in $MDD$ for a region $r$ (dad) and if its position is less than or equal to that stored in $LSRR$ for $r$, then there are no valid delete_attr declarations after the last recovery and so the certificate inserted in $Doc$ is copied by $Doc_{st}$, whereas in set-del-docae is inserted the set of elements declared as deleted associated with $dad$ in $MDD$. Otherwise, if there is a corresponding correct certificate in the currently analyzed document version, then it inserts in $Doc$ this certificate, saves all elements declared as deleted in set-del-docae, and remove from $MDD$ all delete_attr declarations associated with $r$ since a valid delete_attr declaration has been found; else this declaration is left in $MDD$ to be possibly re-evaluated wrt another document version. Point 6 performs more or less the same operations executed in point 5, but wrt update_attr declarations. This point checks if the most recent update_attr declaration for a region is valid wrt the currently analyzed document version according to Definition 20. Point 7 removes from $MDD$ all declarations with position in $F_{pa}$ equal to that of the currently analyzed document version, since no other subsequently required document version can make them valid, and inserts them in $IMDD$. Then it determines the set of most recent declarations in $MDD$ ($decl_{set}$). If at least one declaration is of delete-elem or of other type, but with position greater than that stored in $LSRR$ for the corresponding region, then it sends a request to the subject that has generated this set of declarations to obtain its last stored document version, only in the case in which this subject has not been contacted up to that point. This is done because if this subject was already contacted, then the document version obtained by it would be the previously received one. In the case in which a subject was already contacted or it is unreachable the algorithm remove $decl_{set}$ from $MDD$ and insert $decl_{set}$ in $IMDD$, then it determines the new value of $decl_{set}$ according to the content of $MDD$. When in $decl_{set}$ there is no delete-elem declaration and no delete_attr/update_attr declaration with position greater than that stored in $LSRR$ for the corresponding region, the algorithm removes $decl_{set}$ from $MDD$ and for each declaration in $decl_{set}$ copies content and certificate (case of update_attr declaration) or only certificate (case of delete_attr declaration) in $Doc$ from the stable document version $Doc_{st}$. Finally, when $MDD$ is empty, the algorithm set to null all elements of $Doc$ saved in set-del-docae.
The algorithm ends returning $Doc$, the document recovery version produced, and the corresponding updated IMDD structure. We formally states the correctness of Algorithm 2 in what follows.

**Theorem 11 (Algorithm 2 correctness)** Let $d_c$ be a corrupted document, let $MReg$ be a set of modifiable regions to be recovered, let $IMDD_{st}$ be the last stable version of IMDD structure, let $Doc_{st}$ be the last stable version of the document and let $LSRR_{st}$ be the last stable version of LSRR structure. Algorithm 2 returns the updated $IMDD_{st}$ structure and the last correct region state associated with each region in $MReg$ wrt the modification declarations in $Fpa$, apart from those in the updated $IMDD_{st}$ structure, made valid by the set $DV$ of document versions obtained by contacted subjects in $CG$ and by $Doc_{st}$, according to Definition 22.

### 6.5.5.3 Recovery responses merge algorithm

We now give a description of Algorithm 3 shown in Figure 6.11. Algorithm 3 given $(b+1)$ document recovery versions generates a unique document recovery version ($RecDoc$), composition of the received ones, and a corresponding IMDD structure. Algorithm 3 first (point 1) removes from the set of received $(b+1)$ document recovery versions ($Rec$) those that are not correct wrt their associated IMDD structure. Then, (point 2) Algorithm 3 computes the set of common invalid modification declarations ($CIMDD$) that contains all invalid modification declarations present in all IMDD structures associated with correct document recovery versions. After that it computes the set of valid modification document declarations ($VMDD$). This set contains all delete\_elem modification declarations, associated with regions in $MReg$, present in $Fpa$, but those in $CIMDD$. $VMDD$ also contains for each region $r$ in $MReg$ the delete\_attr modification declaration $dad$ associated with $r$, among the delete\_attr modification declarations in $Fpa$, but those in $CIMDD$, with the highest position in $Fpa$ and the set of atomic elements of $r$ declared as deleted by $dad$ and by other delete\_attr modification declarations in $Fpa$, not in $CIMDD$, with position less than that of $dad$. Furthermore $VMDD$ contains for each region $r$ in $MReg$ the update\_attr modification declaration associated with $r$, among the update\_attr modification declarations in $Fpa$, apart from those in $CIMDD$, with the highest position in $Fpa$. At point 3 the algorithm evaluates each delete\_elem modification declaration $ded$ in $VMDD$ and searches a document recovery version according to which this declaration is valid. Such a document recovery version ($Doc_{ded}$) surely exists, since this declaration has not been inserted in all received IMDD structures. In $RecDoc$ Algorithm 3 inserts the certificate that makes valid $ded$, copied by $Doc_{ded}$, and saves all atomic elements specified in that certificate in component set-del-docae. At point 4 the algorithm determines
Chapter 6: Cooperative updates in colluding byzantine and failure prone distributed systems

Algorithm 3 Recovery Responses Merge (Rec-Resp-Merge)

INPUT: Rec structure: \((\{IMDD_{dl}, Doc_{dl}\})_{dl \in D}\), where \(Doc_{dl}\) is a document, whereas IMDD is the corresponding set of invalid modification document declarations and \(|D| = (b + 1)\)

\(MReg\): set of regions to be recovered

\(RecDoc\): document that will contain the composition of the \((b + 1)\) recovery responses

METHOD:

1. * integrity check of the recovery responses *
   For each \(dl \in D\):
   \[\text{chk} \leftarrow \text{Check}(Doc_{dl}, NMDR(RecDoc), MReg, IMDD_{dl})\]
   If (chk = ERR): \(Rec \leftarrow Rec \setminus \{(Doc_{dl}, IMDD_{dl})\}\)

2. * invalid and valid declarations determination *
   Let CIMDD be the set of common invalid modification document declarations, that is each declaration in CIMDD is invalid for all the recovery responses.

   \(CIMDD = \bigcap_{dl \in D} \text{IMDD}_{dl}\)

   Let VMDD be the set of valid modification document declarations, that is each declaration in VMDD is valid for at least a received recovery response

   \(VMDD \leftarrow \text{ModDecl-Collection}(VMDD, CIMDD, MReg)\)

   \(\text{set-del-docae} \leftarrow \emptyset\)

3. * evaluation of delete_elmt declarations *
   For each \(vded \in \text{VMD-Del-Elmt}\)
   \[\text{For each } Doc_{dl} \in \text{Rec}(Doc_{dl})\]
   \[\text{If } (\exists c \in Doc_{dl}.\text{delete_elmt}_c : (c.obj.root_{c, id} = vded.doc-root_{c, id}) \vee (\exists r \in c.obj.reg[r_{c, id}] : vded.doc-root_{c, id} = c.obj.reg[r_{c, id}].set_{docae}):\]
   \[\text{RecDoc.FDUR}[r].\text{delete}_{elmt-cert} \leftarrow \text{RecDoc.FDUR}[r].\text{delete}_{elmt-cert} \cup \{c\}\]
   \[\text{set-del-docae} \leftarrow \text{set-del-docae} \cup c.obj.reg[r_{c, id}].set_{docae}\]
   \[\text{break}\]

4. * evaluation of delete_attr declarations *
   For each \(r \in MReg\)
   Let \(last_{uad}\) be the \(\text{delete_attr}\) declaration in VMD.Del-Attr associated with \(r\)
   If (\(last_{uad}\) is null): \(RecDoc.FDUR[r].\text{delete}_{attr-cert} \leftarrow \emptyset\)
   Else For each \(Doc_{dl} \in \text{Rec}(Doc_{dl})\):
   \[\text{If } (\exists c \in Doc_{dl}.\text{FDUR}[r].\text{delete}_{attr}_c : (c.obj.pk = last_{uad}.pub\_key)):\]
   \[\text{RecDoc.FDUR}[r].\text{delete}_{attr-cert} \leftarrow \{c\}\]
   \[\text{set-del-docae} \leftarrow \text{set-del-docae} \cup last_{uad}.set_{docae}\]

5. * evaluation of update_attr declarations *
   For each \(r \in MReg\)
   Let \(last_{uad}\) be the \(\text{update_attr}\) declaration in VMD.UP-Attr associated with \(r\)
   If (\(last_{uad}\) is null): \(RecDoc.FDUR[r].\text{update}_{cert} \leftarrow \emptyset\)
   Let \(Doc_{dl}\) be a document in \(\text{Rec}(Doc_{dl})\)
   Insertion in RecDoc of the modifiable content of region \(r\) contained in \(Doc_{dl}\)
   Else For each \(Doc_{dl} \in \text{Rec}(Doc_{dl})\): \(\text{If } (\bigoplus_{r \in \text{current}_{\text{Hash}}(sig_{\text{fu}})} = H(\bigoplus_{r \in \text{current}_{\text{Hash}}(sig_{\text{fu}})})):\)
   \[\text{RecDoc.FDUR}[r].\text{update}_{cert} \leftarrow Doc_{dl}.\text{FDUR}[r].\text{update}_{cert}\]
   Insertion in RecDoc of the modifiable content of region \(r\) contained in \(Doc_{dl}\)
   \[\text{break}\]
   \[\text{break}\]

6. * deletion of content of document atomic elements declared as deleted *
   For each \(docae \in \text{set-del-docae}\)
   Let \(reg\) be the region such that: \(docae \in \text{RecDoc.FDUR[reg].DAE}\)
   \(RecDoc.FDUR[reg].DAE[docae].\text{data} \leftarrow \text{null}\)

\[\text{return } (RecDoc, CIMDD)\]

Figure 6.11: An Algorithm for the merging of recovery responses
for each region \( r \) in \( M_{\text{Reg}} \) the delete\_attr modification declaration \( \text{dad} \) in \( VMDD \) associated with \( r \) and the corresponding document recovery version according to which \( \text{dad} \) is valid, if such a declaration exists. Also in this case a document recovery version \( (\text{Doc}_{\text{dad}}) \) according to which \( \text{dad} \) is valid exists, and the Algorithm 3 inserts in \( \text{RecDoc} \) the certificate that makes valid \( \text{dad} \), and saves in component \( \text{set-del-docae} \) all atomic elements declared as deleted, associated with \( \text{dad} \) in \( VMDD \).

In case there is no delete\_attr declaration the algorithm sets to empty set the component containing delete\_attr certificates for the analyzed region. At point 5 the algorithm determines for each region \( r \) in \( M_{\text{Reg}} \) the update\_attr modification declaration \( \text{uad} \) in \( VMDD \) associated with \( r \) and the corresponding document recovery version according to which \( \text{uad} \) is valid, if such a declaration exists. Also in this case a document recovery version \( (\text{Doc}_{\text{uad}}) \) according to which \( \text{uad} \) is valid exists, and the Algorithm 3 inserts in \( \text{RecDoc} \) the certificate that makes valid \( \text{uad} \), and the content and control information of region \( r \) copied by \( \text{Doc}_{\text{uad}} \). In case there is no update\_attr declaration the algorithm sets to empty set the component containing update\_attr certificates for the analyzed region and insert in \( \text{RecDoc} \) the content and control information of the currently analyzed region copied by a correct document version in \( \text{Rec} \). At the end (point 6) Algorithm 3 sets to null all atomic elements saved in component \( \text{set-del-docae} \) and returns \( \text{RecDoc} \) and the corresponding \( CIMDD \). We formally states the correctness of Algorithm 3 in what follows.

**Theorem 12 (Algorithm 3 correctness)** Algorithm 3, given \((b+1)\) document recovery versions and associated IMDD structures, and the set of modifiable regions to be recovered \((M_{\text{Reg}})\), returns a document recovery version and associated IMDD structure such that each region in \( M_{\text{Reg}} \) contains the last correct region state wrt the modification declarations contained in \( Fpa \) and not present in \( IMDD \), where \( IMDD \) is the set of invalid modification document declarations common to all \((b+1)\) received and elaborated IMDD structures.
Chapter 7

Conclusions and future work
In this thesis we have proposed several approaches to realize distributed and cooperative updates of XML documents. More precisely, we have devised three proposals dealing with this problem, particularly suited for byzantine distributed systems.

All the proposed approaches rely on an access control model specifically tailored for XML documents, allowing to specify access control policies that apply to whole documents or document portions and able to grant to a subject both browsing and authoring privileges on these selected objects. Authoring privileges allow a fine grained level of accuracy in the modification of document content, indeed a subject, according to them, can modify both elements and attributes. Additionally the proposed framework provides an encryption method that, independently encrypting basic portions, called atomic elements, that can be accessed and/or modified, allows subjects to gain access only to the portions for which they possess an access right, according to the stated access control policies.

The first proposed approach deals with cooperative updates in non-colluding byzantine distributed systems, that is there are some subjects involved in the update process that do not obey the stated protocol, but they do not collude. In this approach to the encrypted document is attached an additional control information used by subjects to detect illegal modification operations affecting the document integrity. This control information mainly consists of hash values and digital signatures. This approach also propose a recovery procedure used by the do to build the last correct version of the document content to be sent to the subject that has detected an error during the local document integrity check process. The algorithms proposed in this approach are correct according to the assumption that the subjects do not collude, because an illegal operation executed by a subject and confirmed by a colluding subsequent subject makes it impossible for the other subjects to detect that the content has been corrupted. A complexity analysis of the main operations provided by this approach has also been provided.

The second approach we proposed in this thesis deals with cooperative updates in colluding byzantine distributed systems, that is among the involved subjects there can be some colluding subjects that do not obey the protocol. In this approach the encrypted document cannot be modified during the update process, thus the subjects can only attach their new versions of the document content to the document package. Additional control information is however required to allow subjects to locally execute the document content integrity check. This approach also provide the possibility of specifying semi-dynamic paths, that is a static path is initially stated by the do, then subsequent subjects can extend the path inserting other sub-paths. The protocols provided for this approach are parametric, that is it is possible to fix the maximum number of colluding subjects that do not
affect the protocol. Of course, the amount of information to be exchanged among the subjects grows according to the value specified for the above-mentioned parameter.

The third approach we proposed in this thesis deals with cooperative updates in colluding byzantine failure prone distributed systems, that is not only the protocols are resistant to a number of colluding byzantine subjects, but also they are not affected by a maximum number of failures specified at the beginning. In this respect these are parametric protocols as those presented in the previous approach, but here we have an additional parameter to be taken into account: the number of failures. This approach also provides a language and an infrastructure to specify dynamic paths, called flow policies, that state who must be the subjects that have to receive the document. We provide in this approach the possibility of modifying the specification of these flow policies during the update process according to the stated modifications rules. Another important feature is the recovery process provided by this approach. Recovery process is distributed, that is the last correct version of the document content is built by a set of subjects, called delegates, using the document versions received by the contacted subjects in $CG$. Then a common version of the recovered document must be signed by a number of delegates equal to or greater than the quorum ($Q$) specified by the protocol in terms of the maximum number of colluding byzantine delegates and of the maximum number of down delegates that do not affect the protocol itself. This common recovered document version is accepted by the subject that has required the recovery to the delegates and used as the current document version to be updated and then sent to the next chosen subject.

We retain to extend the work presented in this thesis in several directions.

First we are going to implement in the framework of Author-$\lambda$ the protocols proposed in the last presented approach integrating our proposals with already existing systems. Moreover we are giving a performance evaluation of this implementation in terms of time and space costs.

Second we want to complete the protocols associated with the last proposed approach taking into account the cases in which the Rollback process should be invoked.

Third we are going to develop some mechanisms required to allow a subject to insert in a receiver specification only the minimum information needed to guarantee to the subsequent receivers that it is a valid receiver for a particular position of the flow policy. Moreover we want to take into account the anonymity problem associated with the receivers of the document package.

Finally, we are going to deal with parallel distributed updates of XML documents, that is to allow different subjects to concurrently update disjoint document portions giving also the possibility of generating independent flow policies to be followed by these portions. We believe that a first scenario could be that in which the $do$ states all these independent flow policies at the begin-
ning of the process and communicate this flow policies to the involved parties. Such a flow policies will be static and so non-modifiable during the update process. A possible extension of this schema would be allow subjects to change dynamically these paths preserving all the properties stated for the previous approaches.
Bibliography


[9] E. Fernandez, E. Gudes, and H. Song, ”A Model for Evaluation and Administration of Secu-


[27] SSL (Secure Socket Layer) Available at:  


[34] eXtensible Access Control Markup Language TC. XACML 1.0 Specification Set (18 Feb. 2003): OASIS Standard. Available at: 


http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/

[37] C. Geuer Pollmann, "The XML Security Page", Available at: 
http://www.nue.et-inf.uni-siegen.de/~geuer-pollmann/xmlsecurity.html

[38] World Wide Web Consortium. XML Signature Syntax and Processing, 2002. Available at: 
http://www.w3.org/TR/xmlsig-core/

http://www.w3.org/TR/1999/REC-xpath-19991116

[40] XQuery 1.0: An XML Query Language. Available at: http://www.w3.org/TR/xquery/
Appendix A

Formal Proofs

A.1 Chapter 4

PROOF OF PROPOSITION 1 According to the analysis given above we have that in general the number of sent messages in a conventional centralized system is equal to $2N_i \cdot AR_{avg}$, whereas in our distributed approach is estimated as follows:

$$\#sent\text{-}messages \leq (N_i + 1) + 2(R_R + P_R) + 2P_R \cdot CG + 2R_R \cdot CG$$

$$\leq (N_i + 1) + 2(r_r \cdot N_i + p_r \cdot N_i) + 2p_r \cdot N_i \cdot CG + 2r_r \cdot N_i \cdot CG$$

$$< 2N_i(1 + p_r + r_r + p_r \cdot CG + r_r \cdot CG)$$

It is now clear that when the hypothesis is true the number of sent messages in our distributed approach is less than the number of sent messages required in a conventional centralized system.

PROOF OF PROPOSITION 2 The size of a package for the centralized approach and the size of the set of access requests sent by a subject during a step of a centralized collaborative update process are as follows:

1) $S(CP_d) = S(d^e) + S(\text{digital signature})$.

2) $S(AR) = AE(d) \cdot S(ar) + \sum_{ae \in AE(d)} S(up_{ae})$

By considering that: a) $\exists \bar{c} \in \mathbb{N}: S(d^e) \leq \bar{c} \cdot AE(d)$, because the size of the encrypted document is linear in the number of atomic elements that compose the original document; b) $S(\text{digital signature}), S(ar)$ are constant values; c) $\exists \bar{c} \in \mathbb{N} \forall ae \in AE(d): S(up_{ae}) \leq \bar{c} \cdot S(ae)$; and d) $\exists \bar{c} \in \mathbb{N}: \sum_{ae \in AE(d)} S(ae) \leq \bar{c}AE(d)$, it is clear that exists a natural number $c$ such that:
PROOF OF PROPOSITION 3 The size of a package for the distributed approach is as follows:
\[ S(DP_d) = S(d^*d) + S(id) + R(d) \cdot S(modifiable region) + AE(d) \cdot S(modifiable atomic element) + \text{Path}_d \cdot S(path specification) = S(d^*) + S(id) \cdot [5R(d) + 4AE(d) + 3\text{Path}_d + 1] + S(digital signature) \cdot [4R(d) + AE(d) + \text{Path}_d + 2] + S(hash) \cdot [4R(d) + AE(d)] + 2S(ac) \cdot R(d). \]

By considering that: a) \( \exists \bar{c} \in \mathbb{N} : S(d^*) \leq \bar{c} \cdot AE(d) \), because the size of the encrypted document is linear in the number of atomic elements that compose the original document; b) S(id), S(digital signature), S(hash) and S(ac) are constant values; c) the assumption that R(d) in the worst case has the same cardinality of AE(d); d) the fact that a package contains: 2 digital signatures (one computed over the entire package, and another over the \( H_{NM1} \) control information), and one identifier (component cycle_path); and e) cardinality of \( \text{Path}_d \ll AE(d) \); it is clear that exists a natural number \( c \) such that \( S(DP_d) \leq c \cdot AE(d). \)

PROOF OF PROPOSITION 4 The time required to generate the receiver document view, takes also into account the time required for parsing the document and searching policies that apply to each parsed element. Thus in the above-mentioned worst case: a) the number of policies that apply to the document is equal to AE(d); b) the process used to find out the policy \( p \) that applies to a particular atomic element \( ae \), implies a sequential search in the set \( \mathcal{P} \) that stops when \( p \) is found; and c) the encryption/decryption phases have a cost in time, respectively denoted as \( T(view\ enc) \) and \( T(view\ dec) \), that is proportional to the size of the document, that is \( \exists \bar{c} \in \mathbb{N} : T(view\ enc) \leq \bar{c} \cdot AE(d) \) and \( \exists \bar{c} \in \mathbb{N} : T(view\ dec) \leq \bar{c} \cdot AE(d) \); it is clear that there exists a natural number \( c \) such that:
\[ T(view) = \sum_{i=1}^{AE(d)} i + T(view\ enc) + T(view\ dec) \leq \frac{AE(d) \cdot [AE(d) + 1]}{2} + AE(d) [\bar{c} + \bar{c}] \leq c \cdot (AE(d))^2 \]

PROOF OF PROPOSITION 5 The time required to execute the integrity check protocol applied to \( P_d \), denoted as \( T(P_d) \), can be estimated as follows:
\[ T(P_d) = T(package\ signature\ check) + T(H_{NM1}\ check) + T(\text{Path}_d\ check) + R(d) \cdot [T(step_a) + T(step_b) + T(step_c) + T(step_d) + T(step_e) + T(step_f) + T(step_g)] = T(hash) \cdot [2 + \text{Path}_d + 7R(d)] + T(digital\ signature) \cdot [2 + \text{Path}_d + 3R(d)]. \]

Since in the worst case: a) the number of regions \( R(d) \) is considered equal to the number of the atomic elements \( AE(d) \), b) \( T(hash) \) and \( T(digital\ signature) \) can be considered as constants, c) cardinality of \( \text{Path}_d \ll AE(d) \), d) the decryption time,
denoted as $T(\text{view dec})$, is proportional to the document size, that is $\exists \, \bar{c} \in \mathbb{N}: \, T(\text{view dec}) \leq \bar{c} \cdot \text{AE}(d)$.
e) cardinality of $\text{AccReg}$ is equal to $\text{AE}(d)$, it is clear that exists a natural number $c$ such that:
$T(\text{view}) = T(P_d) + \sum_{i=1}^{\text{AE}(d)} i + T(\text{view dec}) \leq c \cdot [\text{AE}(d)]^2$

A.2 Chapter 6

**PROOF OF PROPOSITION 6**

$op = Q = (2b + d + 1)$ is the solution of the integer linear programming problem that follows:

$$\min op$$

$$\begin{cases} 
    b + \frac{op + d}{2} < Q \\
    op \geq Q 
\end{cases}$$

The first constraint considers the case in which there are $b$ byzantine delegates and no down delegates, thus analyzing the situation in which we have the maximum number of byzantine and a number of operative delegates equal to $(op + d)$. This constraint requires that the sum of the number of signatures produced by the byzantines and by the half of the operative delegates be less than the required quorum. This constraint avoids the situation in which byzantines sign two contents, that are also signed respectively the first by half operative delegates and the second by the other half operative delegates with the aim for the byzantines to have two values signed by at least $Q$ delegates. The second constraint considers the case in which there are $b$ byzantine delegates and $d$ down delegates, thus analyzing the situation in which the number of operative delegates is equal to $op$. The constraint requires that the number of operative delegates be greater or equal than the required quorum, because there are situations in which the protocol requires at least $Q$ signed responses for a request and in presence of $b$ byzantines that can refuse a response and $d$ down delegates that are unreachable, only operative delegates are able to give the required number of signed responses.

**PROOF OF LEMMA 7** We start proving property $a$.

$(a)$: To prove this property we analyze all the sub-steps present in the Delegate Protocol, for a generic step $x$, $x \geq 0$, of the update process, and then we prove that the sub-steps that can be executed, after the execution of the analyzed sub-step, process messages with priority greater than
the priority associated with the message processed by the analyzed sub-step or messages of type *agreement*. We do not analyze sub-step that process message of type *init-dg*, because it is executed only once at step 0 and no more enabled (variable *state* always different by value *initial*). We do not also analyze sub-step that process message of type *end*, because it is executed only once at the end of the update process and it terminates the process itself.

*(sub-step lines 6.7.06-10):* This sub-step processes a message of type *agreement* and with priority equal to 0. Since 0 is the lowest priority and messages of type *agreement* are the only ones associated with this priority, the property holds.

*(sub-step lines 6.7.11-15):* This sub-step processes a message of type *err* and with priority equal to 1. Variable *state* is set to *rec* and variable *requests* has value 0. Enabled sub-steps are those that process messages of types and priorities: (*agreement*, 0), (*fw-rec*, 2), (*fw-signed-fpa-nr*, 4), (*fw-signed-fpa-ar*, 4), and (*end*, 5). Thus, the property holds.

*(sub-step lines 6.7.16-20):* This sub-step processes a message of type *fw-rec* and with priority equal to 2. Variable *state* has value *rec* and variable *requests* is set to value 1. Enabled sub-steps are those that process messages of types and priorities: (*agreement*, 0), (*new-fpa-ar*, 3), (*fw-signed-fpa-nr*, 4), (*fw-signed-fpa-ar*, 4), and (*end*, 5). Thus, the property holds.

*(sub-step lines 6.7.21-25):* This sub-step processes a message of type *new-fpa-nr* and with priority equal to 3. Variable *state* has value *norec* and variable *requests* is set to value 0. Since the execution of this sub-step cause the change of step, from *x* to *x* + 1, the property holds.

*(sub-step lines 6.7.26-30):* This sub-step processes a message of type *new-fpa-ar* and with priority equal to 3. Variable *state* has value *rec* and variable *requests* is set to value 2. Enabled sub-steps are those that process messages of types and priorities: (*agreement*, 0), (*fw-signed-fpa-nr*, 4), (*fw-signed-fpa-ar*, 4), and (*end*, 5). Thus, the property holds.

*(sub-step lines 6.7.21-25):* This sub-step process a message of type *fw-signed-fpa-nr* or *fw-signed-fpa-ar* and with priority equal to 4. Variable *state* is set to value *norec* and variable *requests* is set to value 0. Since the execution of this sub-step cause the change of step, from *x* to *x* + 1, the property holds.

*(b):* To prove this property we assume that in a generic step *x*, *x* ≥ 0, *hpm* stores the higher priority message processed by *dl* ∈ *OP* before going down, and then we analyze the Down Delegate Protocol. Since, by hypothesis, all operative delegates are in step *x* until *dl* becomes operative once again, the components *history*, contained in the received messages of type *agreement*-
resp, do not contain any valid message of type \textit{fw-signed-fpa-nr} or \textit{fw-signed-fpa-ar}. Condition in line 6.8.16 is then satisfied and also that of line 6.8.17. By the For statement of lines 6.8.18-20, all received messages with priority equal to that of \textit{hpm} are removed from \textit{Queue}. \textit{hpm} is surely valid, since the step is the same and the validity is determined regardless the values of variables \textit{state} and \textit{requests}. At line 6.8.21 message \textit{hpm} is inserted in \textit{Queue} whereas line 6.8.22 selects the valid message in \textit{Queue} with the highest priority. This implies that message \textit{m} processed at line 6.8.23 has a priority greater than that associated with \textit{hpm} or \textit{m} = \textit{hpm}. After that, \textit{dl} becomes operative and thus messages chosen to be processed in step \textit{x} satisfy property (a). Property (b) is thus satisfied.

**PROOF OF THEOREM 8** We prove the theorem by induction. The base case is when \textit{x} = 0. In this case, all operative delegates that receive an \textit{init-dg} message make stable the same state \textit{Stat}^0, because we have assumed that the \textit{do} is trusted. Assuming that all delegates have made stable the same state \textit{State}^x, we prove that two generic operative delegates, \textit{dl}, \textit{dlg} ∈ \textit{OP} make stable the same state, \textit{State}^{x+1}. By contradiction, we assume that \textit{dl} makes stable \textit{State}_{dl}^{x+1}, whereas \textit{dlg} makes stable \textit{State}_{dlg}^{x+1}, with \textit{State}_{dl}^{x+1} \neq \textit{State}_{dlg}^{x+1}. This assumption implies that \textit{dl} has processed a message of type \textit{fw-signed-fpa-nr} and \textit{dlg} has processed a message of type \textit{fw-signed-fpa-ar} or viceversa; or they have both processed messages of type \textit{fw-signed-fpa-nr} or \textit{fw-signed-fpa-ar} but with different content. Since each message of type \textit{fw-signed-fpa-nr} must contain \textit{Q} distinct signatures computed on the same message of type \textit{signed-fpa-nr}, then, also in the case in which all \textit{b} byzantines delegates have signed both message of type \textit{signed-fpa-nr} and message of type \textit{signed-fpa-ar}, there are \textit{(b + d + 1)} distinct signatures for message of type \textit{signed-fpa-nr} and \textit{(b + d + 1)} distinct signatures for message of type \textit{signed-fpa-ar} computed by operative delegates. An operative delegate signs a message of type \textit{signed-fpa-nr} or \textit{signed-fpa-ar} when it processes a message of type \textit{new-fpa-nr/new-fpa-ar}. This situation implies that at least one operative delegate has processed two messages with the same priority, but different content, thus contradicting Lemma 7. The thesis is thus proved.

**PROOF OF PROPOSITION 9** According to the protocols a subject, after sending a request to the delegates, waits for a particular number of responses from distinct delegates, that differs according to the request type. In particular, the maximum number of responses from distinct delegates that a subject has to wait for is \textit{Q}, when it sends a message of type \textit{fw-rec}, \textit{new-fpa-nr} or \textit{new-fpa-ar}. In presence of at most \textit{d} down delegates there are however \textit{Q} operative delegates that can send a response. This number is enough also in the case in which there are exactly \textit{b} byzantines delegates
and they send no response. Thus in presence of at most $d$ down delegates, the system goes on with no delay caused by the number of down delegates.

**PROOF OF PROPOSITION 10** According to the Down Delegate Protocol, a down delegate, after sending a message of type *agreement*, waits for $(b + 1)$ responses of type *agreement-resp* from exactly $(b + 1)$ distinct delegates (lines 6.8.08). This number of responses is enough for a down delegate to reach the state common to the other operative delegates and to become operative. According to this situation a down delegate is able to become operative if there are at most $(b + 2d)$ down delegates, because, also in the case in which there are $b$ byzantine delegates that send no response, there are at least $(b + 1)$ operative delegates that send the required response.

**PROOF OF THEOREM 11** We prove this Theorem wrt a generic modifiable region $r \in MReg$. Algorithm 2 requires document versions to be analized in reverse order with respect to the order associated with receiver specification elements in $Fpa$, and it starts from the corrupted document version corresponding to the most recent subject in $Fpa$. This method assures that region states are evaluated starting from the most recent to the last stored in the stable document version. Modification document declarations ($MDD$) to be considered during the evaluation of the analized document versions are those collected by $Rec\text{-MDD-Collection}$ function at the beginning of the algorithm. If a region state belonging to a document version received by a subject in $CG$ satisfies all requirements specified in Definition 22, and thus it is stored (point 6) in $Doc$, then the document recovery version returned by Algorithm 2 surely contains the most recent correct region state, because if another more recent region state satisfies the same conditions, then region $r$ would be no more processed, because all update_attr modification declarations associated with $r$ would have been removed from $MDD$ at the end of point 6. Delete\_\text{elem}t and delete\_\text{attr} modification declarations associated with region $r$ and valid wrt documents received by the contacted subjects or wrt $Doc_{st}$ are evaluated in points 3-4 and 5 respectively. If a delete\_\text{elem}t modification declaration is valid with respect to the current analized document version or with respect to $Doc_{st}$, the set of atomic elements belonging to $r$, to be deleted according to the specification in the certificate that made valid the declaration, are inserted in component *set-del-docae*, that stores all elements declared as deleted by valid declarations. Moreover if a delete\_\text{attr} modification declaration $dad$ is valid with respect to the current analized document version or with respect to $Doc_{st}$, the set of atomic elements belonging to $r$ that are declared as deleted in $dad$ and in those of delete\_\text{attr} present in positions of $Fpa$ less than that associated with $dad$, but not present in $IMDD$, are inserted in component *set-del-docae*. At the end of the algorithm the content of all atomic elements stored in
set-del-docae are set to null. Furthermore, Algorithm 2 determines all valid modification declarations among those initially inserted in MDD, thus assuring that region states inserted in Doc are correct wrt them. Indeed, if a declaration has the same position of the current received document version and this declaration results not valid wrt this document version it is inserted in IMDD and removed from MDD (point 7). If a declaration is not directly evaluated and it is removed from MDD we are in the case in which another more recent declaration of the same type has been made valid by the current document version (removal of declarations from MDD at the end of points 5-6 in presence of a valid update, attr/delete, attr modification declaration).

**PROOF OF THEOREM 12** Document version (RecDoc) returned by Algorithm 3 is surely based on document recovery versions correct wrt their associated IMDD structure, because at the beginning (point 1) all document recovery versions that do not satisfy this requirement are removed by the set of recovery versions to be used to generate RecDoc. Then Algorithm 3 computes the set of common invalid modification declarations (CIMDD) and the set of valid modification document declarations (VMDD). Since RecDoc is built starting from document recovery versions in Rec and declarations in VMDD, it surely contains region states, for regions in MReg, correct wrt declarations in Fpa, but those in CIMDD. Moreover for each region r in MReg, RecDoc contains the last correct r’s state wrt VMDD, because in point 4 and 5 the algorithm finds the document recovery versions that makes valid the most recent update, attr/delete, attr declaration and inserts in RecDoc the corresponding information; moreover in point 3 the algorithm finds, for each delete, element declaration in VMDD, a document recovery version that makes it valid and inserts in RecDoc the corresponding information. A document recovery version that makes valid a generic declaration in VMDD surely exists, since this declaration has not been inserted in all received IMDD structures.