Controlled and cooperative updates of XML documents in byzantine and failure prone distributed systems

GIOVANNI MELLA
University of Milano,
Graal Informatica S.p.a.
ELENA FERRARI
University of Insubria
and
ELISA BERTINO and YUNHUA KOGLIN
Purdue University

This paper proposes an infrastructure and related algorithms for the controlled and cooperative updates of XML documents. Key components of the proposed system are a set of XML-based languages for specifying access control policies and the path that the document must follow during its update. Such path can be fully specified before the update process begins or can be *dynamically* modified by properly authorized subjects while being transmitted. Our approach is fully distributed in that each party involved in the process can verify the correctness of the operations performed till that point on the document without relying on a central authority. More importantly, the recovery procedure also does not need the participation of a central authority. Our approach is based on the use of some special control information that is transmitted together with the document and a suite of protocols. We formally specify the organization of such control information and the protocols. We also analyze security and complexity of the proposed protocols.

Categories and Subject Descriptors: C.2.4 [**Communication Networks**]: Distributed applications; D.4.6 [**Operating Systems**]: Security and Protection—*access control, authentication*; H.2.7 [**Database Management**]: Database Administration—*security, integrity, and protection*

General Terms: Access control, Management, Reliability, Security

Additional Key Words and Phrases: Byzantine problem, distributed systems, languages, updates, XML documents

Author's address: G. Mella, University of Milano, Graal Informatica S.p.a., Italy, email: giovanni_mella@jumpy.it; E. Ferrari, Dipartimento di Scienze della Cultura, Politiche e dell'Informazione, University of Insubria, Como, Italy; email: elena.ferrari@uninsubria.it; E. Bertino, CERIAS and Computer Science Department, Purdue University, USA, email: bertino@cerias.purdue.edu; Y. Koglin, Computer Science Department, Purdue University, USA, email: luy@cs.purdue.edu.

1. INTRODUCTION

The Internet and the development of peer-to-peer systems and standard for workflow languages have made possible a wide spectrum of distributed cooperative applications in several areas, such as collaborative e-commerce [Thuraisingham et al. 2002], distance learning, telemedicine, and 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 document contents must be correct with respect to a given application domain and that they may be modified only by authorized subjects. Another key requirement, besides ensuring document integrity and authenticity, is that documents be actually transmitted to parties according to a specified order, if required by the application at hand. 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 requirement is particularly crucial for novel distributed applications, developed by combining web service technology and workflow systems.

In this paper, we present an approach to the cooperative updates of XML documents particularly suited for byzantine and failure prone distributed systems. With the term *byzantine*, we mean a party involved in the process that does not obey the defined protocols. The basic idea is that the *document originator* (*DO*), that is, the subject[1] that generates the document to be updated and determines who can modify it and according to which mode, sends the document to a given subject; this subject operates on the document and then forwards it to a second subject and so forth. Upon receiving the document from the *DO* or from the previous subject along the path, each subject must be able to modify all and only those portions of the document for which it has a proper authorization, according to the specified access control policies. This goal is obtained through two main components. The first is represented by two high level languages for the specification of *flow policies* and *flow modification rules*, respectively. The language for flow policy specification allows a subject, referred to as *flow policy originator*, possibly different from the *DO*[2], to specify a partial or total path that the document must follow. Further, such a language allows the flow policy originator to directly specify in a flow policy whether a receiver can or cannot extend the flow policy. The language for flow modification is used to specify a set of *flow modification rules* stating which subjects can modify which portions of a flow policy and in which mode.

Based on such languages, we develop an enforcement mechanism able to prevent subjects from executing illegal operations over the flow policies and/or document contents. The mechanism is fully decentralized in that it does not rely on any central authority. In particular, a subject knows which privileges it can exercise

[1]By subject we mean either a human user or a software application.
[2]For simplicity, in the paper, the *DO* is the flow policy originator.

and over which portions of a flow policy or an XML document, because it receives from the originator some certificates attesting the rights it possesses on the flow policy and/or XML document. Upon a modification to a flow policy or to an XML document, the subject has to generate some specialized *control information* to make it possible for a subsequent 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* (Fpa). An XML document together with its corresponding control information forms a *document package*.

We use XML to model both flow policy attachments and document packages, according to a syntax we will explain in the paper. We are thus able to uniformly handle both the XML documents to be protected and the related security and control information. Moreover, the characteristics of XML make it well suited to model the semantics associated with such information, thus making data confidentiality enforcement and data integrity checking more efficient.

The system proposed in this paper considerably extends a preliminary approach developed by Bertino et al. [Bertino et al. 2005]. Such approach has several limitations. It does not provide the languages for specifying the flow policies and the flow policy modification rules. Therefore, it does not support dynamic changes to the flow a document has to follow; this is a serious limitation for applications like workflow systems which may require the path to be dynamically changed after a few workflow tasks have been executed. It assumes that the document recovery is always executed by the *DO* and does not thus support a distributed recovery. It does not account for possible byzantine parties. The current approach addresses all these drawbacks. In particular, here we introduce the possibility of specifying the path that a document must follow and of *modifying* it during the update process. We support such feature through flow paths and related policies for their specification and modification. The key feature of the proposed system is that recovery is fully distributed as the last correct version of a corrupted document is cooperatively built by a set of subjects, called *delegates*. There are many situations under which the *DO* cannot execute the recovery. For example, if there are separation of duty policies, the *DO* may not be allowed to access the document until all modifications to it have been completed. Furthermore, the current approach does not require that all delegates be trusted. It may be extremely difficult to require all delegates trusted, or available throughout the whole update process. Our approach is resilient to byzantine delegates or to failure prone delegates. A byzantine delegate may not follow the protocols (for example, it can build an incorrectly recovered document or avoid sending a response to a request). A failure prone delegate simply is not able to participate to the distributed protocols and does not execute any activity until it joins the system again. To the best of our knowledge, the work reported in this paper is the first to propose such a comprehensive approach to the problem of cooperative updates of XML documents in byzantine and failure prone distributed systems.

To summarize, our current approach achieves the following goals: 1) it allows a subject to view only the document portions for which it has permissions; 2) it allows 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) it allows a subject to detect illegal operations executed over the document content by a single byzantine party or by a set of colluding byzantine parties; and 4) it supports the detection of illegal operations executed over the flow policy by a subject and a fully decentralized recovery of corrupted documents.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 introduces a motivating example. Section 4 introduces the languages used to specify access control and flow policies. Control information used in our protocols is presented in Section 5, whereas an overview of the approach is presented in Section 6. Section 7 describes the protocols we have devised for managing the distributed and cooperative update process. Section 8 presents formal results, whereas 9 deals with recovery. Section 10 evaluates the performance of our approach. Section 11 concludes the paper and outlines future work. Finally, the appendix contains a full description of the protocols presented in Sections 7 and 9.

2. RELATED WORK

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 [Pollmann 1999]. Most of the proposals deal with confidentiality issues and do not consider the problem of controlled document updates. Even though we are not aware of other proposals to which our system can be directly compared, related work includes: proposals concerning the update of XML documents [Tatarinov et al. 2001; Lim et al. 2003; Kane et al. 2002]; group communication techniques and the fault tolerance problem in distributed systems [Vitenberg et al. 1999; Reiter 1996; 1994]; and proposals to manage the illegal behaviour of byzantine subjects [Lamport et al. 1982; Malkhi and Reiter 1997; Malkhi et al. 1999; Malkhi et al. 2001a; Malkhi et al. 2001b]. In general, approaches dealing with updates of XML documents do not consider security issues [Tatarinov et al. 2001; Kane et al. 2002], or rely on centralized approaches [Lim et al. 2003]. Therefore, they are not suitable for highly decentralized environments, as the ones considered in this paper.

The distributed nature of the collaborative update of XML documents presented in this paper implies, as a requirement, the use of group communication techniques. A survey on these techniques are given in [Vitenberg et al. 1999]. Our protocols take into account the fault tolerance problem inherent in the asynchronous and failure-prone nature of distributed systems. Our design has been heavily influenced by protocols proposed in [Reiter 1996; 1994]. The main difference between the previous approaches [Vitenberg et al. 1999; Reiter 1996; 1994] and our is that these approaches are based on the notion of views of a communicating group, according to which messages must be exchanged only between members of the current view. Thus, the communication is stopped whenever the view changes because of the insertion of a new member or the exit of a current one. To overcome this limitation, we adopt a different group management approach which requires to consider, at each instant the initial entire communicating group chosen by the *DO*.

Another feature of our protocols is that they provide methods to mitigate malicious behaviors of a limited number of byzantines. The byzantine problem has been

extensively investigated [Lamport et al. 1982; Malkhi and Reiter 1997; Malkhi et al. 1999; Malkhi et al. 2001a; Malkhi et al. 2001b]. Most of the 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. We borrow from the above mentioned proposals the idea of adding a number of redundant subjects in a communicating group in order 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 a 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 progress.

3. MOTIVATING EXAMPLE

In this section, we provide an example that motivates the need for our infrastructure, cast in the domain of pharmaceutical surveys. In such surveys, doctors from different hospitals are asked to give feedback on several drugs used for common diseases. These drugs are manufactured by different companies. As the survey results will be made public later, these companies would like to see that the feedback favors their drugs.

Participating hospitals are chosen based on an agreement with these companies. The survey document should be circulated among each of the participating hospitals. The order of circulation is fixed before the survey starts. Doctors of a participating hospital will answer the questions asked in the survey document when they receive it. Doctors may also extend the circulation path by adding their nurses, who may update certain sub-sections of the survey for their doctors. Nurses however are not permitted to alter other sub-sections. Only doctors are allowed to extend the circulation path; nurses should not do so.

In order to ensure that the survey is processed correctly, parties such as public notaries are required. These parties are chosen based on the agreement with the companies. If the survey document is corrupted by a malicious participant (for example, in order to favor a certain company, a doctor/nurse may overwrite information on the survey without authorization), the notaries are responsible for recovering the uncorrupted document. We would like to choose the fewest number of notaries possible. However, it is extremely difficult to make all companies to believe that one trusted notary exists which will execute the recovery correctly. In fact, it is possible for a notary to damage the integrity of the survey quite easily. For example, if a doctor extends the document circulation path by letting his nurse fill in parts of it, and if the nurse fills information which does not favor a particular company which patronizes the notary, the notary could delete the information filled by the nurse, as if the doctor did not extend the survey path. Therefore, it is difficult to have a single trusted notary. However, among a number of notaries, we could be confident that a certain number of them would be honest. For example, assume that 80% of the time, any given notary behaves honestly. With 10 notaries public, we expect 8 of them will behave honestly, even though at the beginning of the survey, we are not sure which ones are honest. Also, in reality, some notaries may not *always* be available for monitoring the process due to circumstances be-

```
<Survey name = "Survey.xml" note="Drug Effects">
  <Medicine name = "M1" company = "C1">
       <Doctor name = "Tony" hospital = "H1">
          <Positive> fill in <\Positive>
          <Negative> fill in <\Negative>
          <Num_of_use> N/A  <\Num_of_use>
          <Overall rate = ".." recommend=".."/>
       <\Doctor>
       <Doctor name = "Don" hospital = "H2">
           ....
       <\Doctor>
       ...
  <\Medicine>
  <Medicine name = "M2" company = "C2">
           ....
  <\Medicine>
  ....
<\Survey>
```

Fig. 1. An example of XML document

yond their control. Waiting until every notary becomes available may delay the time for completing the survey, which in turn may delay the involved companies from executing business operations which depends on the result of such surveys (e.g., advertising campaigns).

Another concern is to ensure that the survey is unbiased. Doctors or nurses may be influenced by the answers filled in by people from different hospitals. Thus information provided by doctors or nurses of different hospitals should be kept confidential. Additionally, a doctor or nurse may fill in some information that violates the security and privacy policies of their hospital. Therefore, after they finish answering the survey, the administrative staff of the hospital should check if there are any such violations. If so, they should be able to remove them. However, administrative staff should not access information provided by other hospitals.

4. SPECIFICATION LANGUAGES

Before presenting the specification languages we have developed to support the collaborative and distributed updates of XML document, we first describe the example survey document that will be used throughout the paper.

Example 4.1. The survey document (Figure 1) is for drug effects. For simplicity, the survey concerns drugs M1 to M10 which are manufactured by drug companies C1 to C10, respectively. The subjects who update this document are doctors from hospitals H1 to H50. The doctors should give the positive and negative effects for each drug, the number of times they prescribed it, and the overall efficacy rating of the drug. Doctors may also extend the document flow path by permitting their nurses to update the document.

Access control policies are encoded using the $\mathcal{X}$-sec language [Bertino et al. 2001]. The term *Policy Base* ($\mathcal{PB}$) denotes the XML file encoding access control policies that apply to the *DO*'s XML documents.[3] The *Policy Base* is specified according to the $\mathcal{X}$-Sec *Policy Base* template shown in Figure 2(a). Note that currently, our

[3]We assume that each policy is uniquely identified by an identifier, generated by the system when the policy is specified.

```
<!DOCTYPE policy_base[
  <!ELEMENT policy_base
(policy_spec*)>
  <!ELEMENT policy_spec
EMPTY>
  <!ATTLIST policy_spec
      pid ID
      cred_expr CDATA
#REQUIRED
      target CDATA #REQUIRED
      path CDATA #IMPLIED
      priv (update_attr
           | delete_attr
           | delete_elemt
           | view
           | navigate
           | browse_all)
#REQUIRED
      prop (CASCADE
           | FIRST_LEVEL
           | NO_PROP)
#REQUIRED>
]>
```

(a)

```
<policy_base>
  <policy_spec pid='P1' cred_expr='//Type[@Role='Admin' AND
      host='H1']'
      target='Survey.xml' path='//Doctor[@host='H1']'
      priv='browse_all' prop='CASCADE'/ >
  <policy_spec pid='P2' cred_expr='//Type[@Role='Doctor' AND
      host='H1']' target='Survey.xml'
      path='//Doctor[@host='H1']'
      priv='update' prop='NO_PROP'/ >
  <policy_spec pid='P3' cred_expr='//Type[@Role='Doctor' AND
      host='H2']' target='Survey.xml'
      path='//Doctor[@host='H2']'
      priv='update' prop='CASCADE'/ >
  <policy_spec pid='P4' cred_expr='//Type[@Role='Nurse' AND
      host='H1']' target='Survey.xml'
      path='//Doctor[@host='H1']' priv='update'
      prop='CASCADE'/ >
  <policy_spec pid='P5' cred_expr='//Type[@Role='Admin' AND
      host='H1']' target='Survey.xml'
      path='//Doctor[@host='H1']/Num_of_use'
      priv='delete_elemt' prop='CASCADE'/ >
  ...
</policy_base>
```

(b)

Fig. 2. (a) The $\mathcal{X}$-Sec *Policy Base* template and (b) an example of *Policy Base*

```
<H_staff>
  <Name> Ann  </Name>
  <Eid> 112 </Eid>
  <Type Role="Admin" host ="H1"\>
<\H_staff>
<H_staff>
  <Name> Brian  </Name>
  <Eid> 110 </Eid>
  <Type Role="Doctor" host ="H1"\>
<\H_staff>
```

```
<H_staff>
  <Name> Cathy </Name>
  <Eid> 235 </Eid>
  <Type Role="Nurse" host = "H2"\>
<\H_staff>
<H_staff>
  <Name> Don </Name>
  <Eid> 253 </Eid>
  <Type Role="Doctor" host = "H2"\>
<\H_staff>
```

Fig. 3. Examples of $\mathcal{X}$-Sec credentials

Policy Base template does not support `add_element` and `add_attribute` privileges, because they require a centralized management (cfr. Section 5). We plan to include them in a future version of our protocols.

Example 4.2. Figure 2(b) shows a $\mathcal{PB}$ referring to the XML document in Figure 1. According to the policies in Figure 2(b) administrative employees can browse all information filled by doctors or nurses of their hospital. They can also delete information contained in `Num_of_use` elements for security reason. A doctor or nurse can only update information in the section referring to his/her hospital.

Subjects, to which an access control policy applies, are specified by means of *credentials*, encoded in XML using $\mathcal{X}$-Sec [Bertino et al. 2001]. Examples of $\mathcal{X}$-Sec credentials are presented in Figure 3.

A *flow policy* denotes the sequence of subjects that must receive the document. This sequence can be fully specified 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 contains some *receiver specifications*, that is, properties that have to be verified by the receivers. Each receiver specification

```
<Fpa ...>
  <ReceiverSpec Id = "1">
     <ReceiverProfile Id="2">
        <CredSpec Id = "3">
          //Type[@Role="Doctor" AND @host="H1"]
        </CredSpec>
        <ExtSpec Id="4">subpath</ExtSpec>
     </ReceiverProfile>
  </ReceiverSpec>
  <ReceiverSpec Id = "5">
     <ReceiverProfile Id="6">
        <CredSpec Id = "7">
          //Type[@Role="Admin" AND @host="H1"]
        </CredSpec>
        <ExtSpec Id="8">nosubpath</ExtSpec>
     </ReceiverProfile>
  </ReceiverSpec>
  </ReceiverSpec>
  <ReceiverSpec Id = "9">
     <ReceiverProfile Id="10">
        <CredSpec Id = "11">
          //Type[@Role="Doctor" AND @host="H2"]
        </CredSpec>
        <ExtSpec Id="12">subpath</ExtSpec>
     </ReceiverProfile>
  </ReceiverSpec>
  ...
<\Fpa>
```

Fig. 4. An example of flow policy

contains one or more alternative *receiver profiles*. A receiver 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 [W3C 1999]. Our flow policy specification language enables also an originator to grant a receiver the permission to extend a flow policy by inserting a *sub flow policy*.

Example 4.3. Figure 4 shows a flow policy associated with the document in Figure 1. It specifies that the first receiver must be a doctor in hospital "H1" and the second receiver must be the administrative employee of "H1". Only doctors can extend the flow policy by inserting a new sub flow policy. Thus, a doctor may let his/her nurses update the information.

Modifications to flow policies are governed by flow modification rules, which state which subjects can modify a flow policy. Like credentials and access control policies, flow modification rules are encoded using $\mathcal{X}$-Sec. We denote with the term Rule Base ($\mathcal{RB}$) an XML file encoding a set of flow modification rules. This specification language is very similar to that used to specify access control policies, thus we omit the formal presentation of such a language.

5. CONTROL INFORMATION

In this section, we introduce the control information needed by subjects to check document content integrity, and to correctly exercise their modification rights on the document content. We do not present flow policy control information which is 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 document control information.

Before presenting document control information, we have to introduce some preliminary concepts.

5.1 Preliminary Definitions

Our approach to ensure confidentiality is based on encryption techniques. All the document portions to which the same policies apply are encrypted with the same encryption key. Each subject that has an authorization over some portions of a document receives all and only the keys needed to decrypt those portions. Further details about this encryption method are available in [Bertino and Ferrari 2002]. In particular, the encryption of a document consists of two main phases: the first, referred to as *marking phase*, marks all document portions with a label containing a list of access control policy identifiers, whereas the second, referred to as *encryption phase*, encrypts all document portions according to the strategy explained above.

This leads to the definition of *document atomic element*, which is the basic portion of an XML document that is individually encrypted.

Definition 5.1. **(Document Atomic Element)**. Let d be an XML document. 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[4] a in e: $e.a \in DocAE(d)$;[5] 2) for each element e in d, $e.tags \in DocAE(d)$.

Note that the reason why we may encrypt the start and end tag of an element with a different key wrt the one used for its content and attributes is that we support attribute-level access control policies. Therefore, elements belonging to an XML document d result in two or three non-contiguous atomic components in the encrypted document, depending on their type. By contrast, an attribute always corresponds to a single atomic element (that is, the attribute name and its value, or only the value for data content). For this reason, each encrypted document atomic element *docae* has associated a position information that specifies where *docae*'s components are located in d.

Example 5.2. Examples of atomic elements in the XML document in Figure 1 are:

a) *'name = 'D1"*: the first attribute of the `Doctor` element;

b) *'N/A'*: the data content of the `Num_of_use` element;

c) *'<Overall', '/ >'*: the two components of the empty-element `Overall`;

d) *'<Doctor', '>', '</Doctor>'*: the three components of the start and end tag of `Doctor` element.

The set of atomic elements encrypted with the same key is called a *document region*. We assume that each document region is uniquely identified by an identifier.

The *DO* of an XML document and/or of a flow policy generates a set of signed certificates, containing information concerning the privileges a subject can exercise

[4]For simplicity, we consider the data content associated with an element as an attribute, denoted as "`dc`".

[5]Here and in what follows we use the dot notation to denote a component of a given structure.

over the document and/or flow policy, according to its $\mathcal{PB}$ and $\mathcal{RB}$. Certificates generated for XML documents are called *document modification certificates*, whereas those for flow policies are called *flow policy modification certificates*. These certificates are used by a subject which has modified a document/flow policy portion, to prove its right of modifying that portion to the subsequent receivers of the package.

We do not provide certifcates for `add_element` and `add_attribute` privileges because new inserted nodes should be labeled according to the stated *DO*'s access control policies, thus requiring an additional centralized marking phase.

Definition 5.3. **(Document Modification Certificate)**. Let d be an XML document managed by the *DO* and let $\mathcal{PB}$ be its policy base. Let $Auth\_P(d) \subseteq \mathcal{PB}$ be the set of authoring access control policies that apply to d, and let *acp* be a policy in $Auth\_P(d)$. Let $Sbj\_PK(acp)$ 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 (*cert_id*, *doc_id*, *priv*, *sbj_pk*, *obj*, *signature*), where: *cert_id* is the certificate identifier that univocally identifies a document modification certificate among those generated by the *DO*; *doc_id* is the identifier of d; *priv* is the privilege contained in *acp*; $sbj\_pk \in Sbj\_PK(acp)$; *obj* is one or a set of document regions where *sbj* has privilege *priv* over them according to the *acp*; *signature* is the digital signature of *DO* over the certificate.

Example 5.4. Consider user `Ann`, an administrator belonging to the hospital H1. Consider moreover the document in Figure 1 and the access control policies in Figure 2(b). Furthermore, we assume that: $\mathtt{R}_1$ is the identifier of the document region corresponding to: `//Doctor[hospital =‘‘H1"]/Num_of_use`, whereas $\mathtt{R}_2$ is the region containing the document atomic elements corresponding to `//Doctor[hospital =‘‘H1’’]/Positive`. Then, `(10, Survey, delete_attr,` $\mathtt{PK}_{Ann}$`,` R_1`, signature)`[6] is a valid certificate. Since according to the *DO*'s access control policies, `Ann` is authorized to delete the atomic elements belonging to $\mathtt{R}_1$. By contrast, `(22, Survey, delete_attr,` PK_{Ann}`,` $\mathtt{R}_2$`, signature)` is not a valid certificate, since `Ann` can only view the atomic elements belonging to $\mathtt{R}_2$.

We omit the description of the flow policy modification certificates, since they are very similar to the document modification certificates.

5.2 Document Control Information

The update of the document requires the insertion in the flow policy attachment of some *modification declarations*, having the structure reported in Table I. Moreover, at the end of the document update, s_c must insert in the document, for each modification operation executed on the document, some control information. This guarantees subsequent receivers that s_c 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.

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 of those policies: *non-deletable atomic elements* and *deletable atomic elements*. Since a deletable element

[6] With the notation PK_s we denote the public key associated with subject `s`.

Table I. Modification declaration structure

Notation	Structure	Semantics
`ReceiverSpec`	(..., `DocDecl`, ...)	Single receiver specification information inserted by the corresponding receiver
`DocDecl`	(`Doc-UpAttr-Decl`, `Doc-DelAttr-Decl`, `Doc-DelEl-Decl`)	Modification declaration inserted by a receiver when it modifies the document
`Doc-UpAttr-Decl`	set of *r_id*	document region ids declared as updated
`Doc-DelAttr-Decl`	set of (*r_id*, `del-docae`)	Declaration inserted by the receiver when it deletes some attributes of region *r_id*
`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 of some sub-trees root at *doc-root_id*
`del-reg`	set of *r_id*	set of region ids involved in the deletion

Table II. Control data structures for document atomic elements

Name	Notation	Structure	Semantics
Control structure for non-deletable document atomic elements	*NDAE_LIST*	list of T_{NDAE}, one for each non-deletable document atomic element of *d* belonging to a particular region *r_id*	Control information associated with the non-deletable document atomic elements of *d* belonging to a particular region *r_id*
Control tuple for non-deletable document atomic element	T_{NDAE}	(*docae_id*, *position*, *data*)	Information corresponding to a non deletable atomic element *docae* of a document *d*
Control structure for deletable document atomic elements	*DAE_LIST*	list of T_{DAE}, one for each deletable document atomic element of *d* belonging to a particular region *r_id*	Control information associated with the deletable document atomic elements of *d* belonging to a particular region *r_id*
Control tuple for deletable document atomic element	T_{DAE}	(*docae_id*, *position*, *data*, *h_docae*)	Information corresponding to a deletable document atomic element *docae* of a document *d*

Table III. Components of the control data structures for document atomic elements

Component	Semantics
docae_id	identifier of the document atomic element *docae*
position	value that specifies where *docae*'s components are located in the document
data	encrypted *docae*'s content
h_docae	hash value computed over the *data* component

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_elemt` privilege applies. Table II shows control data structures associated with both the categories, whereas Table III presents the components of the structures introduced in Table II.

Similarly, document regions generated by the document marking can be divided

Table IV. Control data structures for non-modifiable document regions

Name	Notation	Structure	Semantics
Control structure for non-modifiable document regions	NMR	list of T_{NMR}, one for each non-modifiable region of d	Information used by a subject to verify integrity of non-modifiable document regions of d
Control tuple for non-modifiable document regions	T_{NMR}	(r_id, $NDAE\_LIST$, h_nmr_static)	Information corresponding to a specific non-modifiable document region r_id of d

Table V. Components of the control data structures for non-modifiable document regions

Component	Semantics
r_id	identifier of a non-modifiable document region of a document d
h_nmr_static	hash value computed over $NDAE\_LIST$ belonging to r_id: $\mathrm{H}(\sum^{*}_{t \in NMR[r\_id].NDAE\_LIST} t.docae\_id * t.position * t.data)$

Table VI. Modifiable region Classification

Sub-category	Notation	Privileges
Updatable regions	UR	{update_attr}
Partially deletable regions	PDR	{delete_attr}
Fully deletable regions	FDR	{delete_elemt} or {delete_elemt delete_attr}
Partially deletable and updatable regions	$PDUR$	{update_attr, delete_attr}
Fully deletable and updatable regions	$FDUR$	{update_attr, delete_elemt} or {update_attr, delete_elemt delete_attr}

in *non-modifiable* and *modifiable regions*. A region is non-modifiable if all policies that apply to it contain only browsing privileges (i.e., `view`, `navigate`, and `browse_all`); a region is modifiable otherwise. Table IV presents the control data structures for non-modifiable regions, whereas Table V illustrates the semantics of components presented in Table IV. We use character ($*$) to denote the string concatenation operator, whereas we use the notation ($\sum^{*}_{x \in ListX} x$) 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 distinction is made for efficiency purposes. Indeed, in this way, we maximize the amount of content statically protected by specific control information and we also reduce the total amount of control information needed to make possible to check the integrity of a region, thus reducing the time required for the integrity check procedure executed by the protocols.

Table VI 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 {`update_attr, delete_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 the whole set of authoring privileges supported by our model can be exercised. Thus, they represent the most general and complex modifiable region sub-category. According to this assumption, Table VII presents control data structures for $FDUR$ regions only, whereas Table VIII presents the components of the introduced data structures. With reference to Table VIII, the *delete_elmt_cert* component contains the certificates with *delete_elemt* privilege, inserted by subjects when they exercised their modification rights, that apply to disjoint set of atomic elements, thus defined

Table VII. Control data structures for modifiable document regions

Name	Notation	Structure	Semantics
Control structure for document modifiable regions	DMR	(UR, PDR, FDR, $PDUR$, $FDUR$, $delete\_elmt\_cert$)	Information used to verify correctness of document modifiable regions
...	...	...	...
Control structure for fully deletable and updatable regions	$FDUR$	list of T_{FDUR}, one for each fully deletable and updatable region	Information used by a subject to verify integrity of $FDUR$ regions
Control tuple for fully deletable and updatable regions	T_{FDUR}	(r_id, DAE_LIST, h_fdur_static, $sig\_fdudocae$, $update\_cert$, $delete\_attr\_cert$)	Information corresponding to a specific $FDUR$ region r_id

Table VIII. Components of the control data structures for $FDUR$

Component	Meaning and formal specification
$delete\_elmt\_cert$	it contains `non-overlapping` authoring certificates, with `delete_elemt` privilege inserted by the subjects that have executed a deletion over document regions
r_id	identifier of a modifiable document region
h_fdur_static	hash value computed by DO over $docae\_id$, $position$ and h_docae 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 r_id: $H((\sum^{*}_{t \in FDUR[r\_id].DAE\_LIST, type(t)=tags} t.docae\_id * t.position * t.h\_docae) * (\sum^{*}_{t \in FDUR[r\_id].DAE\_LIST, type(t)=attribute} t.docae\_id * t.position))$
$sig\_fdudocae$	digital signature computed over the h_docae component of all the document atomic elements that are attributes listed in DAE_LIST belonging to r_id by the last subject (s_{last}) that has modified the region and whose modification declaration is contained in the receiver specification identified by the information: ($fpa\text{-}id$, ver, $rs\text{-}id$, $orig$), where $fpa\text{-}id$ is a fpa identifier, ver is a fpa version, $rs\text{-}id$ is a receiver specification identifier and $orig$ is a fpa originator $S_{s_{last}}((\sum^{*}_{t \in FDUR[r\_id].DAE\_LIST, type(t)=attribute} t.h\_docae) * fpa\text{-}id * ver * rs\text{-}id * orig)$
$update\_cert$	it contains the authoring certificate with `update_attr` privilege inserted in a region r_id by the last subject that has updated that region
$delete\_attr\_cert$	it contains the authoring certificate with `delete_attr` privilege inserted in a region r_id by the last subject that has deleted at least one attribute of that region

`non-overlapping` certificates.

The signature generated by the last subject that has modified the content of a modifiable region is computed on the components *h_docae* associated with the atomic elements belonging to that region and not on their content (*data* component). Such signature is used to check the integrity of the region. That is, we need to check the integrity of the components *h_docae* and then check the correspondence between each *h_docae* component and the corresponding atomic element content (component data), only for those elements not declared as deleted. The same must be done for modification operations executed on the flow policy.

6. SYSTEM OVERVIEW

Parties involved in the collaborative update process are: a *Cooperative Group*, denoted as $\mathcal{CG}$, a *Delegates Group*, denoted as $\mathcal{DG}$, and the *DO*. Subjects belonging to $\mathcal{CG}$ are chosen by the *DO* at the beginning of the process. They are the only

ones that can be chosen to be the receivers of the XML document. $\mathcal{CG}$ can contain an unlimited number of byzantine subjects. $\mathcal{DG}$ is a set of subjects also chosen by the DO at the beginning of the update process. They are responsible for checking the flow policy integrity (Fpa-Checking) at each step of the process and, whenever required by a subject in $\mathcal{CG}$, to execute document recovery. The set of delegates is partitioned into three subsets: the set of *byzantine delegates* ($\mathcal{B}$, with $|\mathcal{B}| \geq 0$), the set of *operative delegates* ($\mathcal{OP}$, with $|\mathcal{OP}| \geq 0$), and the set of *down delegates* ($\mathcal{D}$, with $|\mathcal{D}| \geq 0$). More precisely, operative delegates obey the protocol and are reachable by the subjects, whereas down delegates are unreachable. A down delegate can become operative again, whereas an operative delegate goes down whenever a failure occurs. Note that, for each delegate in $\mathcal{DG}$, no one can tell whether it belongs to $\mathcal{B}$, $\mathcal{D}$, or $\mathcal{OP}$ at the beginning of the update.

The DO is the subject which generates the XML document (Doc_{DO}) to be updated and the associated flow policy attachment (Fpa_{DO}). This subject also specifies the set of access control policies that apply to Doc_{DO} and the set of flow modification rules that apply to Fpa_{DO}. Before the update process starts, the DO generates and distributes the corresponding document/flow policy modification certificates and document decryption keys to the proper subjects. Decryption keys are needed because we use encryption for confidentiality purposes.

After that, the document generated by the DO is sent to a first chosen subject in $\mathcal{CG}$. Each subject s_c (except the first one) in $\mathcal{CG}$ verifies the document integrity after receiving it, according to the control information in the document it received from the previous subject, and the Fpa it received from the delegates. Whenever an error occurs to the document content, the subject contacts all delegates in $\mathcal{DG}$ to start a recovery. At the end of this recovery, the subject obtains the last correct version of the document and can thus update the document according to its modification rights. During recovery, delegates interact with subjects in $\mathcal{CG}$ to obtain the last correct version of the document and build the recovered document to be sent to the requester.

After subject s_c executes its operations on the document and/or Fpa according to the privileges it possesses, it inserts certificates which can be verified by the later subjects. s_c then sends the Fpa to $\mathcal{DG}$ for Fpa_Checking. That is, s_c should have $\mathcal{Q}$ signatures from delegates which approve the current version of Fpa (as s_c may modify the Fpa). Before s_c sends the document to the next subject, it will make all operative delegates' states stable (this is called Change-Delegates-State). That is, s_c will send to all delegates the Fpa which is signed by at least $\mathcal{Q}$ delegates. Upon receiving the message, each operative delegate will forward this message to other delegates if the message is correct (Fpa is signed by at least $\mathcal{Q}$ delegates). Thus, all operative delegates will have the same Fpa. Also, if s_c requested recovery of the document, it also needs to send the recovered document version which is signed by at least $\mathcal{Q}$ delegates. The purpose of Change-Delegates-State is to ensure that all operative delegates have the same information. That is, they have the same Fpa, recovered document version, etc. Finally, s_c sends the document to the next subject according to the flow policy attachment content. All delegates will send the approved Fpa to the next subject.

After the last subject in the Fpa sends the document to the DO, if the document

is correct, the DO sends a message to end the process. Otherwise, the DO requests the $\mathcal{DG}$ to recover the document in order to get the last correct version of the document.

Before describing our protocols in detail, we present our assumptions of the system and how to set the parameters.

6.1 Assumptions

Our approach relies on a set of assumptions. First, we assume that the DO, each delegate and each subject involved in the update process possesses a private key and that all the other parties know or can retrieve the public key of each other. The DO is in charge of informing, at the beginning, all subjects and delegates of which users compose $\mathcal{CG}$ and $\mathcal{DG}$. Moreover, we assume that there is 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 subject s_c, a *Rollback* procedure is executed to replace s_c with another subject after a fixed amount of time by the last change of state executed by an operative delegate.[7]

6.2 Protocol Parameters Setting

At the beginning of the update process, the DO has to set two parameters: b and d, 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. Parameter b may be set by the system with the default vale of 0, or may be set by the DO. Value d is set as $\lceil c \cdot f \rceil$, where f is an estimated average number of failures proposed by the system and c is the correction parameter set by the DO, the default value of which is 1.

Another important parameter op, which is the number of operative delegates, is strictly related to Quorum $\mathcal{Q}$, which is the minimum number of delegates required for making progress. The relationship between op and $\mathcal{Q}$ is:

$$\begin{aligned} op &\geq \mathcal{Q} && \text{(constraint 1)} \\ b + (op + d)/2 &< \mathcal{Q} && \text{(constraint 2)} \end{aligned}$$

Constraint 1 states that op must be greater than or equal to $\mathcal{Q}$, because in case byzantine delegates do not answer a request, only operative delegates will be able to sign a message content for which the protocol requires at least $\mathcal{Q}$ valid signatures.

Constraint 2 enforces that a byzantine cannot 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, when there is the maximum number of byzantine delegates and no down delegate.[8] The minimum value of op that assures all the above requirements is $(2b + d + 1)$ and the corresponding value for $\mathcal{Q}$ is $(2b + d + 1)$ too.

[7] In the protocols specified in this paper, we do not address this issue, that it is assumed to be a parallel process auditing the delegate behaviour and starting its task when needed.

[8] More details about message types/priorities are presented in Section 7.

7. DISTRIBUTED AND COOPERATIVE UPDATE PROCESS PROTOCOLS

Our approach relies on a suite of protocols, namely: the protocol executed by the *DO* (*Document Originator Protocol*); the protocol executed by the subjects in $\mathcal{CG}$ (*Subject Protocol*); the protocol executed by the operative delegates (*Delegate Protocol*); and, finally, the protocol executed by the down delegates (*Down Delegate Protocol*). In this section we give an high-level description of the protocols, whereas all the details are contained in the appendix (see Figures 9, 10, 11, and 12). Before ilustrating the protocols, we explain some terminology and data structures used in these protocols.

7.1 Terminology and structures

We call $State_{dl}^{x}$ the state associated with a delegate $dl \in \mathcal{OP}$ at step x of the process. We call *step* all the operations/interactions executed by a subject $s_c \in \mathcal{CG}$ 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 ($s_{next} \in \mathcal{CG}$). The complete cooperative and distributed update process thus consists of a set of steps. $State_{dl}^{x}$ which is stored in the local storage of dl, contains the following 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 that indicates when the last recovery occurred for each region ($LSRR$), a vector of progressive numbers used to protect against replay attacks ($N_{\mathcal{IP}}$, where $\mathcal{IP} = \mathcal{CG} \cup \mathcal{DG} \cup \{DO\}$). For a delegate $dl \in \mathcal{OP}$, a step x ends and the subsequent one ($x+1$) begins when dl makes $State_{dl}^{x+1}$ *stable*, 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.

Next, we explain some data structures used by a delegate in more details.

—$IMDD$ (*Invalid Modification Document Declarations*). This structure contains invalid declarations inserted during each recovery (see Example 7.1). The semantics of this structure is presented in Table IX. 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_attr/delete_attr/delete_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.

—$LSRR$ (*Last Saved Region Recovery*). This structure is used by a delegate during the recovery. It 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. During a recovery, only subjects that have declared some modifications on a region to be recovered and that appear in Fpa in a position greater than the one 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.

—$N_{\mathcal{IP}}$. This structure is a vector of progressive numbers, initially sets to all

zeros, one for each party involved in the process. It is used to keep track of the number of steps a subject or the DO has taken part in, and how many times a delegate has requested information in order to be operative.Whenever a subject/DO participates in a step, the corresponding progressive number is increased. The indication of the receiver in the messages sent by a delegate, together with the insertion of the value stored in $N_{\mathcal{IP}}$, which corresponds to that receiver, prevents byzantine subjects and/or byzantine delegates from replaying messages exchanged in a step x during a step y, with $y > x$.

—*state*. This variable contains a string which indicates the state in which the delegate is. For example, the value *norec* for this variable indicates that the delegate has not yet requested a recovery or it is completing a step without the need of a recovery.

—*requests*. This variable is an integer, indicates the number of processed requests wrt the value of variable *state*. If no recovery has been requested, variable *requests* reaches at most value 1, whereas it reaches value 2 in presence of recovery. Variable *state* and *requests* are used to avoid replay attacks within the same step.

—*Queue*. This structure stores all the received messages. During a generic step x, a delegate needs a strategy to choose among all the received messages the next one to process. This strategy is called *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 *Queue* only the messages valid according to $State^x$, and the values of the previous introduced variables. Then, it selects from this set the messages with higher priority and in case of more than one message, the message received first.

Example 7.1. Suppose that user `Don` does not extend the survey path to his nurse `Cathy`. However, `Cathy` maliciously modifies the `rate` attribute filled by `Don` and insertes in the document control information and Fpa her modification declarations. When the document passes to `Lynn`, who is a member of the administrative staff of the hospital, she finds the document corrupted. `Lynn` askes notaries to recover the correct version of the document. Notaries will recover the document by undoing `Cathy`'s modifications, and put `Cathy`'s modification declarations into $IMDD$.

The Subject Protocol also makes use of variable *state* to represent the action the subject is executing or has just executed. A subject also use a structure $N_{\mathcal{CG}}$ which is similar to $N_{\mathcal{IP}}$, to keep track of the number of steps all subjects have taken part in. Whenever a subject s sends a message, this message contains the progressive number associated with s ($N_{\mathcal{CG}}[s]$). This information is used by a receiver to discard old messages.

Parties involved in a cooperative and distributed update process communicate by exchanging messages. Table X gives more details about the exchanged messages. In the table, messages are presented in terms of type, sender, receiver(s), and their complete structure and semantics. Only messages received by a delegate have associated a priority. This is because only Down Delegate and Delegate Protocols use this information to choose the next message to process. Figure 5 shows the

Table IX. IMDD's components

Notation	Structure	Semantics
`IMDD`	(*doc-id*, *doc-version*, *doc-orig*, `sbj-inv-decl`)	structure containing all the invalidate declarations inserted in the flow policy
`sbj-inv-decl`	set of (*fpa-id*, *fpa-version*, *fpa-orig*, *rs-id*, `Up-Attr`, `Del-Attr`, `Del-Elmt`)	set of invalid declarations
`Up-Attr`	set of *r-id*	invalid declaration concerning an update operation over region *r-id*
`Del-Attr`	set of *r-id*	invalid declaration concerning a delete attribute operation over region *r-id*
`Del-Elmt`	set of *doc-root-id*	invalid declaration concerning a delete operation over the sub-tree root at *doc-root-id*

message exchanged between the involved parties.

7.2 DO Protocol

The DO chooses $\mathcal{CG}$ and $\mathcal{DG}$ (line 9.10), and distributes to all delegates information (Doc_{DO}, Fpa_{DO}, $\mathcal{CG}$, $\mathcal{DG}$) (line 9.12).[9] Then, it distributes the decryption keys and document/flow policy modification certificates (lines 9.13-15) to the corresponding subjects in $\mathcal{CG}$. Finally, the DO sends to 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 9.16).

At the end of update, the DO receives from the last receiver subject (sbj) in Fpa a message m containing the document (Doc_{sbj}) and a message signed by $(2b+d+1)$ delegates containing Fpa and $IMDD$ (line 9.17). It checks the document integrity and, if an error occurs, it sends to all delegates an error message for recovery (line 9.21). Each delegate $dl \in \mathcal{DG}$ generates a document recovery version by contacting subjects in $\mathcal{CG}$ and then sends a message containing its version to the DO. The DO accepts the first $(b+1)$ messages from the delegates (line 9.23) and then composes them to obtain the last correct document version (line 9.24).[10] The original document is thus replaced by this new document (line 9.29). At this point, the 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 9.28).

7.3 Subject Protocol

When subject $s_c \in \mathcal{CG}$ receives a message m from a subject $sbj \in \mathcal{CG}$, and at least $b+1$ messages from different delegates such that the structure of Fpa and $IMDD$ are all the same from these messages of the delegates (line 10.07), it can check if m contains a corrupted document according to the received $IMDD$ and Fpa. As we will see in the delegate protocol, an operative delegate will not send Fpa and $IMDD$ to a subject unless at least $(2b+d+1)$ delegates approves this Fpa and $IMDD$.

[9] Here and in the following we reference the lines of the algorithms contained in the Appendix for those readers interested in the details.

[10] More details about the recovery functionality are presented in Section 9.

Table X. Messages

P	Type	Sender	Rcvr(s)	Content and Semantics
-	*init-dg*	DO	$\mathcal{DG}$	(*init-dg*, d_id, Doc_{DO}, Fpa_{DO}, $\mathcal{CG}$, $\mathcal{DG}$): sent by DO to all delegates containing the original version of the document, the initial Fpa and the set of delegates and subjects involved in the process
-	*init-cg*	DO	$\mathcal{CG}$	(*init-cg*, d_id, $regkeys_s, docmodcert_s, fpmodcert_s, \mathcal{CG}, \mathcal{DG}$): sent by DO to each subject s containing s's decryption keys, certificates and the set of subjects and delegates
0	*agreement*	$dl \in \mathcal{D}$	$\mathcal{DG}$	(*agreement*): sent by a down delegate to all delegates to receive information needed to reach the same state of the operative delegates
-	*agreement-resp*	$\mathcal{DG}$	$dl \in \mathcal{D}$	(*agreement-resp, history, hpm, agreements*, $dl, N_{\mathcal{IP}}[dl]$) : sent by delegates to a down delegate containing the history of all previous steps (*history*), their last processed message (*hpm*), all the received but not still processed agreement messages (*agreements*) and information required to prevent other delegates to replay this message (a progressive number and the public key of the down delegate receiver)
1	*err*	$s_c \in \mathcal{CG} \mid DO$	$\mathcal{DG}$	(*err*, $m, S_{sbj}(m), MReg, N_{\mathcal{CG}}[s_c]$): sent by the current subject/DO to all delegates when an error occurs to the document content to collect $(b+1)$ recovery versions
-	*rec*	$\mathcal{DG}$	$s_c \in \mathcal{CG} \mid DO$	(*rec*, $IMDD_{dl_c}, MReg, Doc_{dl_c}, sbj, n_{sbj}$): sent by delegates to the current subject/DO containing the result of their recovery: a Doc and the updated IMDD structure
2	*fw-rec*	$s_c \in \mathcal{CG}$	$\mathcal{DG}$	(*fw-rec*, $\{m, S_{dl}(m)\}_{dl \in D}$): sent by the current subject to all delegates to receive the last correct document version wrt its accessible modifiable regions, obtained unifying the $(b+1)$ forwarded recovery versions
-	*rec-merge*	$\mathcal{DG}$	$s_c \in \mathcal{CG}$	(*rec-merge*, $IMDD_{dl_c}, MReg, Doc_{dl_c}, sbj, n_{sbj}$) sent by delegates to the current subject containing a Doc and the IMDD structure, according to the $(b+1)$ received recoveries
3	*new-fpa-nr*	$s_c \in \mathcal{CG}$	$\mathcal{DG}$	(*new-fpa-nr*,Fpa_{s_c}): sent by the current subject to all delegates to propose a new Fpa, in absence of recovery
-	*signed-fpa-nr*	$\mathcal{DG}$	$s_c \in \mathcal{CG}$	(*signed-fpa-nr*, Fpa): sent by delegates to the current subject if the proposed Fpa is correct, in absence of recovery
3	*new-fpa-ar*	$s_c \in \mathcal{CG}$	$\mathcal{DG}$	(*new-fpa-ar*,Fpa_{s_c}): sent by the current subject to all delegates to propose a new Fpa, after a recovery
-	*signed-fpa-ar*	$\mathcal{DG}$	$s_c \in \mathcal{CG}$	(*signed-fpa-ar*, Fpa): sent by delegates to the current subject if the proposed Fpa is correct, after a recovery
4	*fw-signed-fpa-nr*	$s_c \in \mathcal{CG} \mid \mathcal{DG}$	$\mathcal{DG}$	(*fw-signed-fpa-nr*, $m, \{S_{dl}(m)\}_{dl \in Q}$): sent by the current subject to all delegates and then forwarded by delegates to each other delegate
4	*fw-signed-fpa-ar*	$s_c \in \mathcal{CG} \mid \mathcal{DG}$	$\mathcal{DG}$	(*fw-signed-fpa-ar*, $\overline{m}, \{S_{dl}(\overline{m})\}_{dl \in \overline{Q}}, \widehat{m}, \{S_{dl}(\widehat{m})\}_{dl \in \widehat{Q}}$) sent by the current subject to all delegates and then forwarded by delegates to each other
5	*end*	DO	$\mathcal{DG} \cup \mathcal{CG}$	(*end*, d_id): end the update process

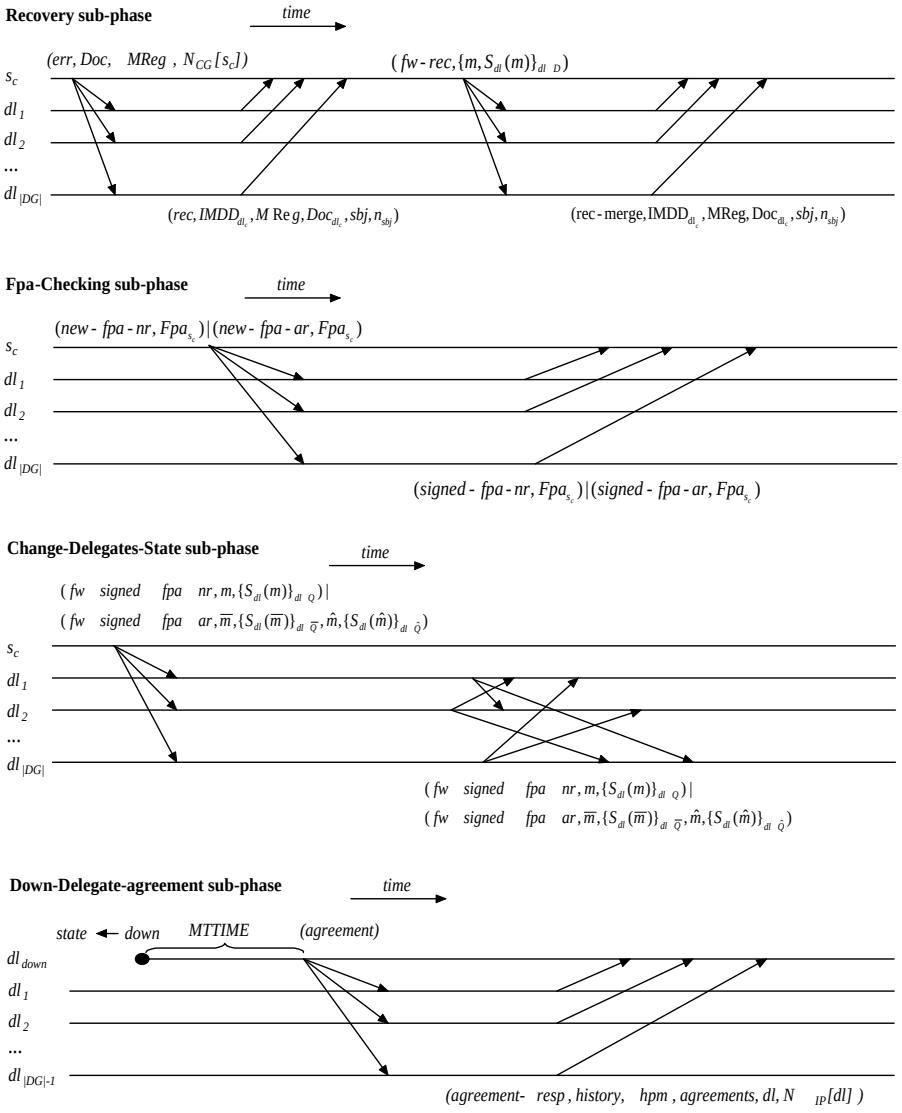

Fig. 5. messages exchange

If there is no error in m, s_c executes the operation on the document (line 10.21). Otherwise (line 10.09), it sends to all delegates a recovery message (line 10.12) to obtain the last correct document version wrt the set of regions it can access.

s_c accepts the recovery reply messages from $(b+1)$ delegates (line 10.14) and then puts these messages in a message $\overline{m}$ and sends $\overline{m}$ to all delegates (line 10.16). Next, s_c waits to receive recovery messages from at least $(2b+d+1)$ delegates. The recovery results from these $(2b+d+1)$ delegates must be the same (line 10.18). Then this finishes the recovery and s_c can start operations on the correct document

version.

After executing the operation on the document and/or Fpa (line 10.21-22), s_c has to send its updated Fpa to all delegates in order to be checked and signed (This is called Fpa_Checking). If no recovery has been requested by s_c during the step, s_c sends a message (*new-fpa-nr*,Fpa_{s_c}) to all delegates (line 10.25); a message (*new-fpa-ar*,Fpa_{s_c}) is sent otherwise (line 10.28). s_c has to send to each delegate a message whose type depends on whether it has requested a recovery or not in the step. A delegate verifies if the updated Fpa is correct, that is, if s_c has the certificates which authorize it to update the Fpa and it inserted these certificates in the Fpa. If so, the delegate signs the message containing the Fpa and sends it back.

Example 7.2. When `Don` extends the flow path by letting his nurse `Cathy` to update the survey, he should get enough signatures (at least of $\mathcal{Q}$ delegates) by requesting Fpa_Checking for the new proposed flow path. Fpa_Checking is also important to prevent byzantine delegates from damaging the integrity of the updates. As previously mentioned in Section 3, if a byzantine delegate deletes what `Cathy` filled in because the information does not favor the drug company he favors, he must have a message from `Don` who proposed the flow path without extension. However, he could not get such message from `Don` since it contradicts `Don` operations.

When s_c receives $(2b+d+1)$ such signatures from delegates, it is ready to change delegate state. That is, before sending the document to the next subject, s_c has to notify all delegates the document recovery version and corresponding $IMDD$ structure generated during the recovery, if any, and the correct Fpa proposed. s_c does this by sending a message $\widetilde{m}$ to all delegate. If no recovery happened in the step, $\widetilde{m}$ is of type *fw-signed-fpa-nr* (line 10.31) and it contains the new Fpa and $(2b+d+1)$ signatures of it from delegates. Otherwise, $\widetilde{m}$ is of type of *fw-signed-fpa-ar* (line 10.37) and it contains: 1) message $\overline{m}$ of type *fw-signed-fpa-ar* (line 10.35) containing the new Fpa; 2) $(2b + d + 1)$ delegate signatures on $\overline{m}$; 3) message $\widehat{m}$ of type *rec-merge* containing the document recovery version and associated IMDD structure; 4) $(2b + d + 1)$ delegate signatures computed on $\widehat{m}$.

After s_c sends $\widetilde{m}$, it sends the document to the next subject according to the Fpa (line 10.40-43). s_c needs to wait until receiving a message from the DO (line 10.44) indicating that the update process ends. Before the update process end, some delegates may contact s_c for recovering the document.

As we will see in the delegate protocol, a subject cannot generate a valid message of type *fw-signed-fpa-nr* and another valid message of type *fw-signed-fpa-ar* in the same step, because the protocols prevent this subject from collecting $\mathcal{Q}$ signatures for a message of type *new-fpa-nr* and $\mathcal{Q}$ signatures for a message of type *new-fpa-ar*. Indeed, in the same step, a delegate does not accept messages with the same priority.

Example 7.3. After `Don` extends the path and gets Fpa_Checking, the document is passed to `Cathy`. `Cathy` fills in parts of the survey, according to the specified access control policies. Later, a byzantine delegate cannot undo `Cathy`'s update, even by colluding with `Don`, since at least $\mathcal{Q}$ delegates should sign Fpa when `Don`

first proposed it. Don cannot get $\mathcal{Q}$ delegates to sign a different Fpa indicating that he did not extend the path, since an operative delegate does not sign different messages of the same priority (in this case, two different Fpas) in one step.

7.4 Delegate Protocol

The initial *state* of a delegate is *norec*. When an operative delegate receives a message of type *agreement* from a down delegate (line 11.06), it replies with a response message $\widetilde{m}$ which contains the information from which the down delegate can reach the same state of the operative delegates (see Section 7.5).

If a delegate *dl* receives a recovery message (line 11.11) from *sbj* who is in $\mathcal{CG}$, *dl* checks the following before doing any recovery: 1) according to the Fpa stored in the current state, *sbj* should be the subject doing the update process now. As we will show later, all operative delegates always have the same and correct Fpa at each step, due to the *State* consistancy; 2) the request associated with the current step is 0; 3) current *state* variable is *norec*; 4) the document contained in the recovery message is signed by a previous subject who is before *sbj* in the Fpa; 5) this is not a replay attack, according to the information stored in $N_{\mathcal{IP}}$. If there is any error, *dl* just ignores the message. Otherwise, it generates a document recovery version and the corresponding $IMDD$ updated structure by contacting subjects in $\mathcal{CG}$ and then sends to *sbj* a message containing the generated information.

If *dl* receives from *sbj* a message m of type *fw-rec* (line 11.16), it will check the following: 1) variable *state* is *rec*; 2) $sbj \in \mathcal{CG}$; 3) variable *requests* is 0; 4) according to the Fpa, *sbj* is the current subject who is updating the document; 5) there are $(b+1)$ recovery messages signed by different delegates. If there is no error, *dl* sets $requests = 1$ and m as the *hpm*; then it generates a merge version of *Doc* and $IMDD$ (line 11.18) and sends it to *sbj* (line 11.20).

When *dl* receives a message m from $sbj \in \mathcal{CG}$ for Fpa_Checking (line 11.21 for no recovery situation and line 11.26 for recovery situation), it checks the following. If the message m is of type *new-fpa-nr*, *state* variable must be *norec* and *requests* must be equal to 0, as this is the first request from *sbj* in this step. If the message m is of type *new-fpa-ar*, *state* variable must be *rec* and *requests* must be equal to 1. Also, from the Fpa stored in *dl*'s *State*, *sbj* should be the current subject requesting for Fpa_Checking. If all above are satisfied, *dl* increases the variable *requests* by 1 and set m as the *hpm*. *dl* then checks the integrity of the proposed Fpa from *sbj*. If no error in the proposed Fpa, *dl* sends a signed message of type *signed-fpa-nr* (line 11.24) or type *signed-fpa-nr* (line 11.29) to *sbj*, depending whether a recovery happened or not in this step.

When *dl* receives a message m of type *fw-signed-fpa-nr* or *fw-signed-fpa-ar* from *sbj* for commit the step (line 11.31), it checks the following. 1) In the case that m is of type *fw-signed-fpa-nr*, them m must contain a message $\overline{m}$ of type *signed-fpa-nr* and $(2b+d+1)$ delegate signatures on $\overline{m}$. This indicates that at least $(2b+d+1)$ delegates agree with the Fpa proposed in $\overline{m}$. 2) In the case that m is of type *fw-signed-fpa-ar*, them m must contain a message $\overline{m}$ of type *signed-fpa-ar* and $(2b+d+1)$ delegate signatures on $\overline{m}$. Also, m must contain a message $\widetilde{m}$ of type *rec-merge* which contains a recovered version of document and the corresponding $IMDD$, and $(2b+d+1)$ delegate signatures on $\widetilde{m}$. If the above are satisfied and the message is not a replay attack according to the information stored in N_{IP} , *dl*

updates the components of *State* (line 11.32), that is, it sets the variable *state* as *norec*, $requests = 0$, $hpm = m$ and puts m into *history*. *dl* also sends the received message m to other delegates (line 11.33), in case *sbj* did not send m to them. So all the operative delegates will have the same state. That is, in case m of type *fw-signed-fpa-ar*, all operative delegates set components of their *State*, such as the current document version, $IMDD$, $LSRR$ etc. according to the one contained in $\widetilde{m}$. *dl* thus finishes a step, and *State* is stable. It then sends a message containing Fpa and $IMDD$ to the next subject in the path (line 11.39).

The designed protocols assure that each operative delegate signs only once such information at each step, thus preventing byzantine subjects from obtaining two different contents signed by at least $\mathcal{Q}$ distinct delegates and forcing different delegates, such as $dl_i, dl_j \in \mathcal{OP}$, to make stable $State_{dl_i}^{x+1}$ and $State_{dl_j}^{x+1}$ with $State_{dl_i}^{x+1} \neq State_{dl_j}^{x+1}$.

7.5 Down Delegate Protocol

A down delegate dl_{down} that wakes up at time t in step x, waits for a time equal to $MTTIME$ (line 12.04) before sending a message (*agreement*) to all delegates. This allows messages sent before t and not received by dl_{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_{down}.

An operative delegate sends a response message m = (*agreement-resp, history, hpm, agreements,*$dl_{down}, N_{\mathcal{IP}}[dl_{down}]$), where: *history* contains all the messages of type *fw-signed-fpa-nr* or *fw-signed-fpa-ar* received by an operative delegate from the beginning of the process, *hpm* is the last accepted and processed highest priority message, *agreements* contains all the received and not yet processed messages of type *agreement*. $m.dl_{down}$ and $m.N_{\mathcal{IP}}[dl_{down}]$ are information used to avoid a replay attack. dl_{down} accepts the first $(b + 1)$ received responses coming exactly from $(b + 1)$ distinct delegates (line 12.08). After that, dl_{down} processes all valid messages in the $(b + 1)$ received *history* components, and makes stable the most recent state which is the same with at least one delegate whose response has been accepted by dl_{down}. That is, dl_{down} processes messages of type *fw-signed-fpa-nr* or *fw-signed-fpa-ar* until structure *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, dl_{down} processes the message in *Queue* chosen according to the received message scheduling policy, here applied with the variant of no taking into account variables *state* and *requests*.

After that, the Down Delegate protocol ends and dl_{down} becomes operative.

8. FORMAL RESULTS

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

The following lemma and the subsequent theorem states that our protocols assure that the state made stable by two different operative delegates is the same, regardless the malicious behavior of at most b byzantine delegates and the presence of some down delegates.

LEMMA 8.1. **(Message Execution Uniqueness)** *Let* $\mathrm{dl} \in \mathcal{OP}$. *The Delegate and Down Delegate protocols assure the following properties:*

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

Proof. (Sketch). a): To prove this property, we analyze all the sub-steps of the Delegate Protocol and show that after the execution of each sub-step, only higher priority messages (except *agreement* messages) are executed. We do not analyze the sub-step that processes message of type *init-dg*, because it is executed only once at step 0 and no more enabled (variable *state* is always different by value *initial*). Neither do we analyze the sub-step that processes 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 11.06-10): 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 11.11-15): This sub-step processes a message of type *err* and with priority equal to 1. Variable *state* is set to *rec* and variable *request* 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 11.16-20): This sub-step processes a message of type *fw-rec* and with priority equal to 2. Variable *state* has value *rec* and variable *requestes* 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 11.21-25): 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 1. 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 11.26-30): This sub-step processes a message of type *new-fpa-ar* and 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 11.21-25): This sub-step processes messages of type *fw-signed-fpa-nr* or *fw-signed-fpa-ar* and with priority equal to 4. Variable *state* is set to *norec* and variable *requests* is set to 0. Since the execution of this sub-step causes a change of step, from x to $x + 1$, the property holds.

b): To prove this property, we assume that in a generic step x, $x \geq 0$, *hpm* stores the higher priority message processed by $dl \in \mathcal{OP}$ 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 *history* components contained in the received messages of type *agreement-resp*, do not contain any valid message of type *fw-signed-fpa-nr* or *fw-signed-fpa-ar*. Condition in line 12.16 is then satisfied and also that of line 12.17. By the For statement in lines 12.18-20, all received messages with priority equal to that of *hpm* are removed from *Queue*. *hpm* is surely valid, since the step is the same and the validity is determined regardless the values of variables *state* and *requests*. At line 12.21 message *hpm* is inserted in *Queue*, whereas line 12.22 selects the valid message in *Queue* with the highest priority. This implies that message m processed at line 12.23 has a priority greater than that associated with *hpm*, or $m = hpm$. After that, dl becomes operative and thus messages chosen to be processed in step x satisfy property (a). Property (b) is thus satisfied. □

The next theorem indicates that all operative delegates keep the same state.

THEOREM 8.2. **(Delegate State Uniqueness)** *Let* dl, dlg $\in \mathcal{OP}$. *If* dl *makes* $State^x_{dl}$ *stable, and* dlg *makes* $State^x_{dlg}$ *stable, then* $State^x_{dl} = State^x_{dlg}$, *for* x ≥ 0.

PROOF. (Sketch). The theorem could be proved by induction. The base case is when $x = 0$. In this case, all operative delegates that receive an *init-dg* message make stable the same state $State^0$, because we have assumed that the DO is trusted. Assuming that all delegates have made stable the same state $State^x$, we prove that two generic operative delegates, $dl, dlg \in \mathcal{OP}$ make stable the same state, $State^{x+1}$. By contradiction, we assume that dl makes stable $State^{x+1}_{dl}$, whereas dlg makes stable $State^{x+1}_{dlg}$ such that $State^{x+1}_{dl} \neq State^{x+1}_{dlg}$. This assumption implies that dl has processed a message of type *fw-signed-fpa-nr* and dlg has processed a message of type *fw-signed-fpa-ar* or viceversa; or they have both processed messages of type *fw-signed-fpa-nr* or *fw-signed-fpa-ar* but with different content. Each message of type *fw-signed-fpa-nr*/*fw-signed-fpa-ar* must contain $(2b + d + 1)$ distinct signatures computed on the same message of type *signed-fpa-nr*/*signed-fpa-ar*. Assume that all b byzantine delegates have signed both message of type *signed-fpa-nr* and message of type *signed-fpa-ar*, there are still left $(b + d + 1)$ distinct signatures for message of type *signed-fpa-nr* and $(b + d + 1)$ distinct signatures for message of type *signed-fpa-ar* computed by other delegates. Assume there is no down delegates, then there is at least one operative delegate who has processed two messages with the same priority, but different content, thus contradicting Lemma 2. Thus, the thesis holds. □

The following propositions respectively state that our protocols are not delayed by the presence of a number of down delegates less than or equal to d, that is, regardless of the presence of delegates not reachable or not enabled to accomplish their tasks, the system continues the execution without delay. They also state that the system is able to make progress also in presence of a number of down delegates less than or equal to $(b + 2d)$, assuring the liveness of the update process also in this adversary condition.

PROPOSITION 8.3. *(***Delay freeness***) The maximum number of down delegates that do not delay the system is* d.

PROOF. (Sketch). According to the protocols, after sending a request to the delegates, a subject 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 $\mathcal{Q}$, when it sends a message of type *fw-rec, new-fpa-nr* or *new-fpa-ar*. In presence of at most d down delegates there are however $\mathcal{Q}$ operative delegates that can send a response. This number is enough also in the case in which there are exactly b byzantine 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. □

PROPOSITION 8.4. *(***System survivability***) The system survives, that is, it is 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).

PROOF. (Sketch). 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. 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. □

9. RECOVERY

The goal of recovery is to retrieve the last correct version of the regions accessible by s_c. It accomplishes this task by contacting subjects in $\mathcal{CG}$ that have received till that point at least a document package and that have executed at least a modification operation on at least one region accessible by s_c. Also, it is desirable to limit as much as possible the number of subjects to be contacted. Correctness of a region is determined according to the set of valid modification declarations inserted in Fpa.

A declaration of deletion of a document sub-tree is valid if: 1) a certificate corresponding to this declaration exists, or 2) a certificate corresponding to a deletion declaration of a document sub-tree that includes the deleted sub-tree exists.

A `delete_attr` modification declaration is valid if the subject that has inserted this declaration in Fpa has also inserted in the document control information of the currently analyzed document version a corresponding certificate. A `delete_attr` modification declaration is also valid if it is not evaluated as invalid till that point and another valid subsequent `delete_attr` declaration exists in Fpa. That is, a `delete_attr` declaration is considered valid if a subsequent subject s will correctly exercise the `delete_attr` privilege on the same region. The same strategy applies to `update_attr` and `delete_elemt` modification declarations. Even more, a valid declaration for the deletion of a sub-tree a, and its corresponding certificate, makes other not yet evaluated deletions of a sub-tree b such that $b \subseteq a$, valid. This implies that a recovery is done from backwards.

Example 9.1. When `Don` receives the survey document, he first performs the integrity checking of the received document. He finds that the `rate` attribute of the `Doctor` element which he should fill in has been illegally updated by `Paul`, who

also inserted `update_attr` declaration in the document control information without a valid certificate. `Don` could simply update the attribute without requesting any recovery. Later on, when `Cathy` needs to update the survey document (as Don extended the flow path) and finds that the survey document is corrupted by someone during the document transmission, she requests a recovery. In this case, `Paul`'s modification is considered valid. Since Don's valid update overwrote Paul's invalid update. It does not affect the integrity of the survey.

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.

Next, we give a high level description of the recovery presented in the Appendix (Figures 13 and 14). This algorithm is used by a delegate to generate a document recovery version and the corresponding $IMDD$ structure.

Initially, the algorithm replaces all non-modifiable information in the document to be recovered (Doc), with those in the stable version of the document (Doc_{st}). Then function `Rec-MDD-Collection` collects in the MDD data structure all modification declarations not yet inserted in the stable version of $IMDD$ and associated with at least a region in $MReg$. Each `delete_attr/update_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 with which the modification declaration is associated. Each `delete_attr` modification declaration (*dad*) in MDD associated with a region r contains: 1) 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*. For each region in $MReg$, the most recent valid `update_attr/delete_attr` declarations determined during the last recovery is inserted in MDD.

The algorithm considers the position in Fpa corresponding to the subject (`sbj`(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_elemt/delete_attr` declarations. Algorithm 1 analyzes declarations in MDD until the set is empty and when required it contacts a subject to obtain a document version against which to evaluate the declarations in MDD. Algorithm 1 starts by the document version to be recovered and then, if required, other document versions are obtained in reverse order wrt the order associated with receiver specifications in Fpa.

When the algorithm analyzes certificates associated with `delete_elemt` privileges, it removes all certificates that are not correct, or that are not associated with a `delete_elemt` declaration, or that are contained in another certificate. `Delete_elemt` declarations are considered valid according to the strategy we described at the beginning of this section. Elements declared as deleted in these valid declarations are saved in *set-del-docae* component.

Similarly, the algorithm 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*). 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. Therefore, the certificate inserted in Doc is copied by Doc_{st}, also inserting in *set-del-docae* the set of elements declared as deleted associated with *dad* in MDD. 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 removes from MDD all `delete_attr` declarations associated with r. At point 5, the algorithm analyzes `update_attr` declarations. Next, at point 7 all declarations with position in the Fpa equal to that of the currently analyzed document version are removed from MDD, since no other subsequently required document version can make them valid. They are inserted in $IMDD$.

Then, the algorithm determines the set *decl_set* which contains the declarations of the subject who is at the highest position in the Fpa among the subjects who made declarations in MDD. If at least one declaration is at a position greater than the one stored in $LSRR$ for the corresponding region, then a request is sent to the subject that has generated this set of declarations to obtain its last stored document version, if this subject has not been contacted up to that point. If the subject is unreachable, the algorithm removes *decl_set* from MDD and inserts *decl_set* in $IMDD$, then it determines the new value of *decl_set* according to the content of MDD. When *decl_set* does not contain anymore `delete_elemt` declarations and `delete_attr/update_attr` declarations with position greater than the one stored in $LSRR$ for the corresponding region, the algorithm removes *decl_set* from MDD and for each declaration in *decl_set* it copies content and certificate (in case of `update_attr` declarations) or only certificate (in case of `delete_attr` declarations) in Doc from the stable document version Doc_{st}.

Finally, when MDD is empty, the algorithm sets to *null* all elements of Doc saved in *set-del-docae*. The algorithm ends returning Doc, the produced document recovery version, and the corresponding updated $IMDD$ structure.

The correctness of the algorithm can be found in [Mella 2004]. Given $(b + 1)$ document recovery versions, all delegates perform the same merge operation. That is, they first check the integrity of the received versions and delete these invalid ones. Then, from these valid versions, for each modifiable region, they find the most recent (according to the Fpa) update as the final version for this region. Therefore, all operative delegates will generate the same version at the end. They sign such one and send it to the requester. A recovery responses merge algorithm can be found in [Mella 2004]).

10. PERFORMANCE EVALUATION

This section evaluates the performance of our approach. We compare it with the case that only one party (trusted) is involved for flow path integrity checking and document recovery.

10.1 Experimental Setup

We measure the time to complete one *step*. Both delegates (or the trusted party) and subjects ran on identical Solaris workstations. These workstations have 450MHz CPUs and 2GB of memory. The network bandwidth is 100MB/s. Point-to-point

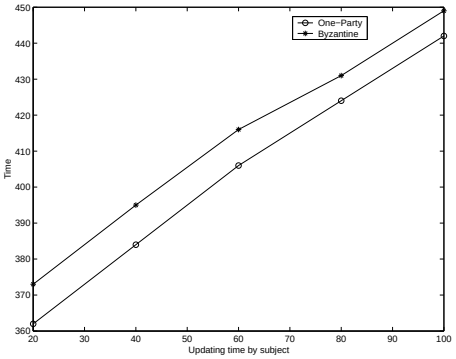

Fig. 6. Time (no recovery)

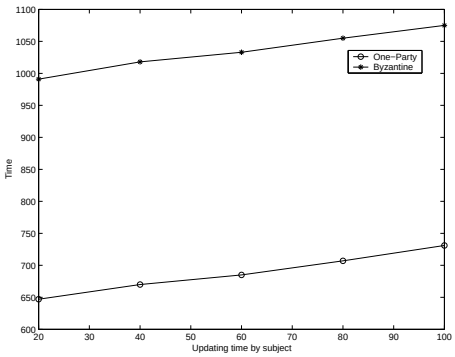

Fig. 7. Time (with recovery)

communications is implemented using TCP. RSA of 1024-bit modulus is used for digital signatures. To ensure the confidentiality of the document, AES with a key size of 256 bit is used for encryption.

We performed the experiment with 1 byzantine delegate and 0 down delegates. Totally, there are 4 delegates, 3 of them operative. Fpa is about 5KB and the document is 100KB.

10.2 Results

Figure 6 reports the time complexity when no recovery is requested. In this case, our approach does not have much more delay than the one-party approach. The overhead introduced in our approach results from the time spent on the change-delegate-state sub-phase. This sub-phase requires the subject to collect 3 signatures on the proposed Fpa and then to broadcast them to all delegates, which takes less than 10ms. This low overhead is due to the fact that the subject does not need to perform any encryption or generate any signature; it only needs to verify at most 4 signatures. Compared to the overall time (362ms) of the one-party case when the operation time is 20ms, this overhead is only 3%. When operation time increases, this overhead decreases.

Next, we measured the time complexity when recovery is requested. Figure 7 reports the results when only one subject needs to be contacted by delegates (or one-party) for recovery of the document regions which is of size 0.5KB. As we can see, the overall time increases for both approaches. Even for the one-party case, the time is almost double that the time with no recovery. This is due to the fact that the party needs to decrypt the document, and then to find the subjects to be contacted in order to recover the document. Encryption/decryption are needed for the communication between the subjects and the delegates (one-party). This process also requires the signatures computation and verification, and transferring the recovered version to the requested subject. Thus, recovery almost doubles the time needed for both approaches. However, the byzantine approach has more overhead than the one-party case. For example, when the operation time is 20 ms, the byzantine approach needs 991ms while one party needs 647ms. The overhead is almost 35%.

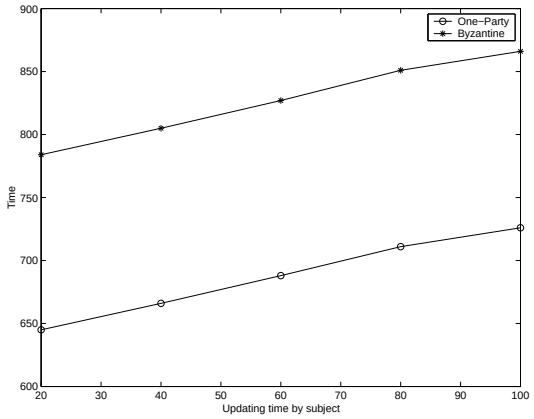

Fig. 8. Time (with recovery)

This overhead is mainly due to the encryption and decryption of recovered versions for recovery-merge operations and also to the time required to delegates to send back the merged version. Furthermore, the subject needs to make delegates states stable by sending the merged version and signatures on it. We optimized the performance by reducing the size of messages for recovery merge. If the first $b + 1$ recovered versions are the same, only one copy is sent to the delegates, along with all the digital signatures of the version. Similarly, a delegate only needs to send back the digital signature on the version that it approves. Figure 8 shows the results. The overhead is now reduced to only 18%.

11. CONCLUSION AND FUTURE WORK

In this paper, we have proposed an approach supporting cooperative updates in byzantine and failure prone distributed systems. The protocols we have developed are resistant to a number of colluding byzantine subjects, and they are not affected by a maximum number of failures specified at the beginning. We have also developed a language and an infrastructure to specify dynamic paths, called flow policies, stating which subjects have to receive the document. In particular, we provide 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 as part of our approach. Indeed, recovery 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 $\mathcal{CG}$.

We plan to extend the work presented in this paper in several directions. First, we want to develop protocols to manage rollback. Second, we plan to extend our work with mechanisms supporting receiver anonymity, and `add_element` and `add_attribute` privileges. Third, we will investigate an efficient key mamagement for encryption/decryption documents. We also plan to relax the constraint that subjects in $\mathcal{CG}$ cannot be changed once the update starts, which is currently a limitation of our approach. Finally, we will deal with how to allow different subjects

to concurrently update disjoint XML document portions in a parallel distributed setting [Koglin et al. 2005], giving the possibility of generating independent flow policies to be used on these portions.

REFERENCES

Bertino, E., Castano, S., and Ferrari, E. 2001. On specifying security policies for web documents with an xml-based language. In *Proceedings of the 1st ACM Symposium on Access Control Models and Technologies*. ACM, Chantilly, VA, USA, 49–59.

Bertino, E. and Ferrari, E. 2002. Secure and selective dissemination of xml documents. *ACM Transactions on Information and System Security (TISSEC) 5,* 3, 290–331.

Bertino, E., Ferrari, E., and Mella, G. 2005. An approach to cooperative updates of xml documents in distributed systems. *Journal of Computer Security 13,* 2, 191–242.

Kane, B., Su, H., and Rundensteiner, E. 2002. Consistently updating xml documents using incremental constraint check queries. In *Proceedings of the 4th ACM CIKM International Workshop on Web Information and Data Management (WIDM'02)*. ACM, Virginia, USA, 1–8.

Koglin, Y., Mella, G., Bertino, E., and Ferrari, E. 2005. An update protocol for xml documents in distributed and cooperative systems. In *Proceedings of the 25th International Conference on Distributed Computing Systems*. ACM, Ohio, USA, 49–59.

Lamport, L., Shostak, R., and Pease, M. 1982. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems 4,* 3, 382–401.

Lim, C., Park, S., and Son, S. 2003. Access control of xml documents considering update operations. In *Proceedings of the ACM Workshop on XML Security*. ACM, Virginia, USA, 49–59.

Malkhi, D., Mansour, Y., and Reiter, M. K. 1999. On diffusing updates in a byzantine environment. In *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*. EEE Computer Society, Lausanne, Switzerland, 134–143.

Malkhi, D. and Reiter, M. K. 1997. Byzantine quorum systems. In *Proceedings of the 29th ACM Symposium on Theory of Computing*. ACM, El Paso, Texas, 569–578.

Malkhi, D., Reiter, M. K., Rodeh, O., and Sella, Y. 2001a. Efficient update diffusion in byzantine environments. In *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems*. IEEE Computer Society, New Orleans, USA, 90–98.

Malkhi, D., Reiter, M. K., Wool, A., and Wright, R. N. 2001b. Probabilistic quorum systems. *The Information and Computation Journal 170,* 2, 184–206.

Mella, G. 2004. Distributed and cooperative updates of xml documents. Ph.D. thesis, University of Milano, DICO Department, Milano, Italy. Availabe at: http://homes.dico.unimi.it/dbandsec/mellagiovanni.

Pollmann, C. G. 1999. The xml security page. Availabe at: http://www.nue.et-inf.uni-siegen.de/ẽeuer-pollmann/xml_security.html.

Reiter, M. K. 1994. Secure agreement protocols: Reliable and atomic group multicast in rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*. ACM, Fairfax, Virginia, USA, 68–80.

Reiter, M. K. 1996. A secure group membership protocol. *IEEE Transactions on Software Engineering 22,* 1, 31–42.

Tatarinov, I., Ives, Z. G., Halevy, A. Y., and Weld, D. S. 2001. Updating xml. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*. ACM, California, USA, 413–424.

Thuraisingham, B., Gupta, A., Bertino, E., and Ferrari, E. 2002. Collaborative commerce and knowledge management. *Knowledge and Process Management 9,* 1, 43–53.

Vitenberg, R., Keidar, I., Chockler, G., and Dolev, D. 1999. Group communication specifications: A comprehensive study. In *Tech. report CS9931*. Comp. Sci. Inst., The Hebrew University of Jerusalem and MIT Technical Report MIT-LCS-TR-790.

W3C. 1999. XML Path Language (Xpath) 1.0. Availabe at: http://www.w3.org/TR/1999/REC-xpath-19991116.

APPENDIX

Protocols presented informally in Section 7 are presented formally in Figures 9, 10 11, 12 in this appendix. To better understand protocols we first introduce details about the semantics of some notations. The boolean function **rcvd**() used in the algorithms returns true, if the message specified as its argument has been received by the subject that executes the protocol, false otherwise. In some cases, it is also required the specification of the sender as function parameter, that is, the function returns true if the specified message has been received by the specified sender. The semantics for:

repeat
[]$C_1 : A_1$
[]$C_2 : A_2$
...
[]$C_n : A_n$

is that the following procedure is repeated: all the conditions are evaluated and among the conditions that are evaluated as true, if there are more than one, the condition C_k which is chosen is the one that takes into account the message with the highest priority. Accordingly A_k is executed and a new evaluation of the condition is performed at the end of the execution of A_k or when the time assigned to the process that execute A_k expires. If a condition C_i different by C_k is chosen and the process that executes A_k has not terminated its execution, a new process is started to execute A_i and the process associated with A_k is killed. By contrast if C_k is again chosen for the same previous message the process associated with A_k continues its execution. Evaluating conditions and executing statements are repeated until the **repeat** statement is terminated with a **terminate repeat** statement. A **terminate repeat** statement terminates only the closest **repeat** statement. Symbol ($\leftarrow$) denotes the assignment operator. A single assignment assumes the following form: $a \leftarrow v_a$, where v_a is the value to be assigned to variable a. A multiple assignment assumes the following form: $(a, b) \leftarrow (v_a, v_b)$, where v_a is the value to be assigned to variable a, and v_b is the value to be assigned to variable b.

Table XI summarizes the functions invoked in the protocols.

Received Month Year; revised Month Year; accepted Month Year

Table XI. Functions

Function	Semantics
`Full`	return *true* if Fpa has been completed (there is no free element), else *false*
`Last_rcv`	return the public key of the subject that has proposed Fpa as current version (last subject registered in Fpa).
`Ass_flow`	return the identifier of Fpa associated with Doc
`Id`	return the identifier associated with Doc or Fpa
`Check`	check the document (Doc) integrity according to IMDD structure and wrt the set of non-modifiable ($NMReg$) and modifiable ($MReg$) regions
`Rec-Resp-Merge`	insert in Doc the last correct version of regions in $MReg$, using region versions and invalid declarations produced by (b+1) delegates
`Last_next`	find the public key of the subject chosen by the last receiver to be the next receiver
`Update-Doc`	subject modifies Doc according to the rights it possesses
`Update-Fpa`	subject modifies Fpa according to the rights it possesses, insert its information in Fpa, and choose the next receiver (s_{next})
`MDR / NMDR`	extract modifiable/ non-modifiable regions from a document or a structure containing regions and keys
`Recovery`	it collects last correct modifiable regions content asking to the subjects, that has modified these regions, their last generated region versions
`Rcvrs`	return the number of receivers that has already received Fpa
`Check-Fpa`	check the integrity of Fpa
`Stable`	replace previous state at step x ($State^x$) with the subsequent state ($State^{x+1}$)
`Clock`	return the value of the internal clock
`Extract-msg`	insert in $Queue$ messages of type *fw-signed-fpa-nr* or *fw-signed-fpa-ar* contained in $history$ structures received during an agreement phase
`Received-msg`	add to $Queue$ messages received after the last execution of this function
`Higher-priority`	find in structure $Queue$ the message with higher priority that satisfies a delegate condition statement, regardless the values for $state$ and $requests$
`Valid`	return *true* if the analyzed message satisfies at least one delegate condition statement, regardless the values for $state$ and $requests$, *false* otherwise
`Priority`	return the priority of message according to the priorities specified in Tables X
`Type`	return the type of the message according to types specified in Tables X
`Execute`	execute 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: $state \leftarrow norec$, $requests \leftarrow 0$

(9.01) $state \leftarrow initial$
(9.02) Let Doc_{DO} be the document to be updated and Fpa_{DO} the associated flow policy attachment
both chosen by the DO
(9.03) Let $NMReg$ and $MReg$ be the set of non-modifiable/modifiable regions associated with Doc_{DO}
(9.04) Let d_id be the identifier of Doc_{DO}
(9.05) **repeat**
(9.06) [] $state = initial$:
(9.07) Generation of the package to be sent to the first chosen subject, $P \leftarrow (s_1, Doc_{DO}, Fpa_{DO})$
(9.08) Set parameters b and d to the value chosen by the document originator (DO)
(9.09) Choice of subjects in $\mathcal{CG}$ and delegates in $\mathcal{DG}$.
(9.10) $m \leftarrow$ (*init-dg*, d_id, Doc_{DO}, Fpa_{DO}, $\mathcal{CG}$, $\mathcal{DG}$)
(9.11) **send** $(\langle m, S_{DO}(m)\rangle)$ to $\mathcal{DG}$
(9.12) **For each** $s \in \mathcal{CG}$:
(9.13) $\overline{m} \leftarrow$ (*init-cg*, d_id, $regkeys_s, docmodcert_s, fpmodcert_s, \mathcal{CG}, \mathcal{DG}, s$)
(9.14) **send** $(\langle \overline{m}, S_{DO}(\overline{m})\rangle)$ to s
(9.15) **send** $(\langle P, S_{DO}(P)\rangle)$ to s_1
(9.16) $state \leftarrow final$
(9.17) [] $state = final \wedge \exists\ sbj \in \mathcal{CG}(\mathbf{rcvd}(sbj, \langle m, S_{sbj}(m)\rangle) \wedge m = (Doc_{sbj}, n_{sbj}) \wedge \exists\ Q \subset \mathcal{DG}$ $(\mathbf{rcvd}(\langle m, \{S_{dlg}(m)\}_{dlg \in Q}\rangle) \wedge m = (Fpa, IMDD, n) \wedge |Q| = 2b + d + 1) \wedge (n = n_{sbj}) \wedge$ $\texttt{Full}(Fpa) \wedge (\texttt{Last\_rcv}(Fpa) = sbj) \wedge (\texttt{Ass\_flow}(Doc_{sbj}) = \texttt{Id}(Fpa)) \wedge (\texttt{Id}(Doc_{sbj}) =$ d_id):
(9.18) **If** $(\texttt{Check}(Doc_{sbj}, NMReg, MReg, IMDD) = \texttt{ERR})$:
(9.19) $state \leftarrow recovery$
(9.20) $\overline{m} \leftarrow (err, m, S_{sbj}(m), MReg, 0)$
(9.21) **send** $(\langle \overline{m}, S_{DO}(\overline{m})\rangle)$ to $\mathcal{DG}$
(9.22) **else** $state \leftarrow end$
(9.23) [] $state = recovery \wedge \exists D \subset \mathcal{DG}(\forall dl \in D(\mathbf{rcvd}(\langle m, S_{dl}(m)\rangle) \wedge (\texttt{Id}(Doc_{dl}) = \text{d\_id}) \wedge$ $m = (rec, IMDD_{dl}, MReg, Doc_{dl}, DO, 0) \wedge |D| = b + 1)$:
(9.24) $(Doc, IMDD) \leftarrow \texttt{Rec-Resp-Union}(\{(IMDD_{dl}, Doc_{dl})\}_{dl \in D}, MReg, Doc_{DO})$
(9.25) $state \leftarrow end$
(9.26) [] $state = end$:
(9.27) $m \leftarrow$ (*end*, d_id)
(9.28) **send** $(\langle m, S_{DO}(m)\rangle)$ to $(\mathcal{CG} \cup \mathcal{DG})$
(9.29) Replace Doc_{DO} with Doc in the DO XML source
(9.30) **terminate repeat**

Fig. 9. Document Originator Protocol

(10.01) $(state, first) \leftarrow (initial, 0)$
(10.02) **repeat**
(10.03) [] $state = initial \wedge$ **rcvd**$(sbj, \langle m, S_{sbj}(m)\rangle) \wedge$
$m = (init\text{-}cg,$ d_id, $rk_{s_c}, docmodcert_{s_c}, fpmodcert_{s_c}, \mathcal{CG}, \mathcal{DG}, s_c)$:
(10.04) $(state, MReg, NMReg, N_{\mathcal{CG}}, DO) \leftarrow (wait, MDR(rk_{s_c}), NMDR(rk_{s_c}), [0 \ldots 0]_{\mathcal{CG}}, sbj)$
(10.05) [] $state = wait \wedge$ **rcvd**$(\langle m, S_{sbj}(m)\rangle) \wedge sbj = DO \wedge m = (s_c, Doc, Fpa) \wedge$
$(first = 0) \wedge$ (Id(Doc) = d_id):
(10.06) $(state, first) \leftarrow (update, 1)$
(10.07) [] $state = wait \wedge \exists\ sbj \in \mathcal{CG}($**rcvd**$(\langle m, S_{sbj}(m)\rangle) \wedge m = (Doc, n_{sbj})) \wedge$
$\exists D \subset \mathcal{DG}(\forall dl \in D($**rcvd**$(\langle \overline{m}, S_{dl}(\overline{m})\rangle) \wedge \overline{m} = (Fpa, IMDD, n_{sbj})) \wedge |D| = b + 1) \wedge$
$(N_{\mathcal{CG}}[sbj] \leq n_{sbj}) \wedge$ (Last_rcv$(Fpa) = sbj) \wedge$ (Last_next$(Fpa) = s_c) \wedge$
(Ass_flow(Doc) = Id$(Fpa)) \wedge$ (Id(Doc) = d_id):
(10.08) $N_{\mathcal{CG}}[sbj] \leftarrow (n_{sbj} + 1)$
(10.09) **If** (Check$(Doc, NMReg, MReg, IMDD)$ = ERR):
(10.10) $state \leftarrow recovery$
(10.11) $\overline{m} \leftarrow (err,\ m, S_{sbj}(m), MReg, N_{\mathcal{CG}}[s_c])$
(10.12) **send** $(\langle \overline{m}, S_{DO}(\overline{m})\rangle)$ to $\mathcal{DG}$
(10.13) **else** $state \leftarrow update$
(10.14) [] $state = recovery \wedge \exists D \subset \mathcal{DG}(\forall dl \in D($**rcvd**$(\langle m, S_{dl}(m)\rangle) \wedge$ (Id(Doc_{dl}) = d_id))
$\wedge\ m = (rec,\ IMDD_{dl}, MReg, Doc_{dl}, s_c, N_{\mathcal{CG}}[s_c])) \wedge |D| = b + 1)$:
(10.15) $\overline{m} = (fw\text{-}rec,\ \{m, S_{dl}(m)\}_{dl \in D})$
(10.16) **send**$(\langle \overline{m}, S_{s_c}(\overline{m})\rangle)$ to $\mathcal{DG}$
(10.17) $state \leftarrow fw\text{-}rec$
(10.18) [] $state = fw\text{-}rec \wedge$
$\exists Q \subset \mathcal{DG}(\forall dl \in Q($**rcvd**$(\langle m, S_{dl}(m)\rangle) \wedge$ (Id(Doc) = d_id) $\wedge$
$m = (rec\text{-}merge, IMDD, MReg, Doc, s_c, N_{\mathcal{CG}}[s_c])) \wedge |Q| = 2b + d + 1)$:
(10.19) $state \leftarrow update\text{-}ar$
(10.20) [] $(state = update \vee state = update\text{-}ar)$:
(10.21) $Doc_{s_c} \leftarrow$ Update-Doc(Doc)
(10.22) $(Fpa_{s_c}, s_{next}) \leftarrow$ Update-Fpa(Fpa)
(10.23) **If** $(state = update)$:
(10.24) $m \leftarrow (new\text{-}fpa\text{-}nr, Fpa_{s_c})$
(10.25) **send**$(\langle m, S_{s_c}(m)\rangle)$ to $\mathcal{DG}$
(10.26) $state \leftarrow sent\text{-}fpa\text{-}nr$
(10.27) **else** $m \leftarrow (new\text{-}fpa\text{-}ar, Fpa_{s_c})$
(10.28) **send**$(\langle m, S_{s_c}(m)\rangle)$ to $\mathcal{DG}$
(10.29) $state \leftarrow sent\text{-}fpa\text{-}ar$
(10.30) [] $state = sent\text{-}fpa\text{-}nr \wedge$
$\exists Q \subset \mathcal{DG}(\forall dl \in Q($**rcvd**$(\langle m, S_{dl}(m)\rangle) \wedge m = (signed\text{-}fpa\text{-}nr, Fpa_{s_c})) \wedge |Q| = 2b+d+1)$:
(10.31) $\widetilde{m} \leftarrow (fw\text{-}signed\text{-}fpa\text{-}nr,\ m, \{S_{dl}(m)\}_{dl \in Q})$
(10.32) **send**$(\langle \widetilde{m}, S_{s_c}(\widetilde{m})\rangle)$ to $\mathcal{DG}$
(10.33) $state \leftarrow doc\text{-}delivery$
(10.34) [] $state = sent\text{-}fpa\text{-}ar \wedge$
$\exists Q \subset \mathcal{DG}(\forall dl \in Q($**rcvd**$(\langle m, S_{dl}(m)\rangle) \wedge m = (signed\text{-}fpa\text{-}ar, Fpa_{s_c})) \wedge |Q| = 2b+d+1)$:
(10.35) $\overline{m} \leftarrow (signed\text{-}fpa\text{-}ar,\ Fpa)$
(10.36) $\widehat{m} \leftarrow (rec\text{-}merge,\ IMDD, MReg, Doc, s_c, N_{\mathcal{CG}}[s_c])$
(10.37) $\widetilde{m} \leftarrow (fw\text{-}signed\text{-}fpa\text{-}ar,\ \overline{m}, \{S_{dl}(\overline{m})\}_{dl \in \overline{Q}}, \widehat{m}, \{S_{dl}(\widehat{m})\}_{dl \in \widehat{Q}})$
(10.38) **send**$(\langle \widetilde{m}, S_{s_c}(\widetilde{m})\rangle)$ to $\mathcal{DG}$
(10.39) $state \leftarrow doc\text{-}delivery$
(10.40) [] $state = doc\text{-}delivery$:
(10.41) $\overline{m} \leftarrow (Doc_{s_c}, N_{\mathcal{CG}}[s_c])$
(10.42) **send**$(\langle \overline{m}, S_{s_c}(\overline{m})\rangle)$ to s_{next}
(10.43) $(N_{\mathcal{CG}}[s_c], state) \leftarrow ((N_{\mathcal{CG}}[s_c] + 1), wait)$
(10.44) [] $state = wait \wedge$ **rcvd**$(\langle m, S_{sbj}(m)\rangle) \wedge m = (end,$ d_id$) \wedge sbj = DO$:
(10.45) **terminate repeat**

Fig. 10. Subject Protocol

(11.01) $(state, requests, IMDD_{st}, history) \leftarrow (initial, 0, \emptyset, \emptyset)$
(11.02) **repeat**
(11.03) [] $state = initial \wedge$ **rcvd**$(sbj, \langle m, S_{sbj}(m)\rangle) \wedge$
$m = (init\text{-}dg,$ d_id, $Doc_{st}, Fpa_{st}, \mathcal{CG}, \mathcal{DG}) \wedge (dl_c \in \mathcal{DG})$:
(11.04) $(state, \mathcal{IP}, N_{\mathcal{IP}}, DO) \leftarrow (norec, \mathcal{CG} \cup \mathcal{DG} \cup \{DO\}, [0 \ldots 0]_{\mathcal{IP}}, sbj)$
(11.05) $LSRR \leftarrow [(\texttt{Id}(Fpa_{st}), 0) \ldots (\texttt{Id}(Fpa_{st}), 0)]_{MDR(Doc_{st})}$
(11.06) [] $(state = norec \vee state = rec) \wedge \exists\ dl \in \mathcal{DG}($**rcvd**$(\langle m, S_{dl}(m)\rangle)) \wedge m = (agreement)$:
(11.07) Let $agreements$ be the set of not still processed messages of type $agreement$ present in $Queue$
(11.08) $\widetilde{m} \leftarrow (agreement\text{-}resp, history, hpm, agreements, dl, N_{\mathcal{IP}}[dl])$
(11.09) **send**$(\langle \widetilde{m}, S_{dl_c}(\widetilde{m})\rangle)$ to dl
(11.10) $N_{\mathcal{IP}}[dl] \leftarrow N_{\mathcal{IP}}[dl] + 1$
(11.11) [] $state = norec \wedge \exists\ sbj, \widehat{sbj} \in \mathcal{CG}($**rcvd**$(\langle m, S_{sbj}(m)\rangle)) \wedge$
$m = (err, \widehat{m}, S_{\widehat{sbj}}(\widehat{m}), MReg, n_{sbj})) \wedge (\texttt{Last\_next}(Fpa_{st}) = sbj) \wedge (N_{\mathcal{IP}}[sbj] = n_{sbj})$
$\wedge\ \widehat{m} = (Doc, n_{\widehat{sbj}}) \wedge \widehat{sbj} = \texttt{Last\_rcv}(Fpa) \wedge N_{\mathcal{IP}}[\widehat{sbj}] = n_{\widehat{sbj}} \wedge (requests = 0)$:
(11.12) $(state, hpm, IMDD_{rec}) \leftarrow (rec, \langle m, S_{sbj}(m)\rangle, IMDD_{st})$
(11.13) $(Doc_{dl_c}, IMDD_{dl_c}) \leftarrow \texttt{Recovery}(Doc, MReg, IMDD_{rec}, Doc_{st}, LSRR)$
(11.14) $\overline{m} \leftarrow (rec, IMDD_{dl_c}, MReg, Doc_{dl_c}, sbj, n_{sbj})$
(11.15) **send**$(\langle \overline{m}, S_{dl_c}(\overline{m})\rangle)$ to sbj
(11.16) [] $(state = rec) \wedge \exists\ sbj \in \mathcal{CG}, D \subset \mathcal{DG}($**rcvd**$(\langle m, S_{sbj}(m)\rangle) \wedge |D| = (b+1) \wedge$
$m = (fw\text{-}rec, \{\overline{m}, S_{dl}(\overline{m})\}_{dl \in D})) \wedge \overline{m} = (rec, IMDD_{dl}, MReg, Doc_{dl}, sbj, n_{sbj})$
$\wedge\ (\texttt{Last\_next}(Fpa_{st}) = sbj) \wedge (N_{\mathcal{IP}}[sbj] = n_{sbj}) \wedge (requests = 0)$:
(11.17) $(requests, state, Rec\text{-}Doc, hpm) \leftarrow (1, rec, Doc_{st}, \langle m, S_{sbj}(m)\rangle)$
(11.18) $(Doc_{dl_c}, IMDD_{dl_c}) \leftarrow \texttt{Rec-Resp-Merge}(\{(IMDD_{dl}, Doc_{dl})\}_{dl \in D}, MReg, RecDoc)$
(11.19) $\overline{m} \leftarrow (rec\text{-}merge, IMDD_{dl_c}, MReg, Doc_{dl_c}, sbj, n_{sbj})$
(11.20) **send**$(\langle \overline{m}, S_{dl_c}(\overline{m})\rangle)$ to sbj
(11.21) [] $state = norec \wedge \exists\ sbj \in \mathcal{CG}($**rcvd**$(\langle m, S_{sbj}(m)\rangle) \wedge m = (new\text{-}fpa\text{-}nr,\ Fpa)) \wedge$
$(\texttt{Last\_next}(Fpa_{st}) = sbj) \wedge (\texttt{Rcvrs}(Fpa) = \texttt{Rcvrs}(Fpa_{st}) + 1)$
$(requests = 0)$:
(11.22) $(requests, hpm, state) \leftarrow (1, \langle m, S_{sbj}(m)\rangle, norec)$
(11.23) **If** $(\texttt{Check-Fpa}(Fpa)) \neq \texttt{ERR})$:
(11.24) $\overline{m} \leftarrow (signed\text{-}fpa\text{-}nr,\ Fpa)$
(11.25) **send**$(\langle \overline{m}, S_{dl_c}(\overline{m})\rangle)$ to sbj
(11.26) [] $state = rec \wedge \exists\ sbj \in \mathcal{CG}($**rcvd**$(\langle m, S_{sbj}(m)\rangle) \wedge m = (new\text{-}fpa\text{-}ar,\ Fpa)) \wedge$
$(\texttt{Last\_next}(Fpa_{st}) = sbj) \wedge (\texttt{Rcvrs}(Fpa) = \texttt{Rcvrs}(Fpa_{st}) + 1)$
$(requests = 1)$:
(11.27) $(requests, hpm, state) \leftarrow (2, \langle m, S_{sbj}(m)\rangle, rec)$
(11.28) **If** $(\texttt{Check-Fpa}(Fpa)) \neq \texttt{ERR})$:
(11.29) $\overline{m} \leftarrow (signed\text{-}fpa\text{-}ar,\ Fpa)$
(11.30) **send**$(\langle \overline{m}, S_{dl_c}(\overline{m})\rangle)$ to sbj
(11.31) [] $(state = norec \vee state = rec) \wedge (\exists\ sbj \in \mathcal{CG}, Q \subset \mathcal{DG}($ **rcvd**$(\langle m, S_{sbj}(m)\rangle) \wedge$
$|Q| = (2b + d + 1) \wedge m = (fw\text{-}signed\text{-}fpa\text{-}nr, \overline{m}, \{S_{dlg}(\overline{m})\}_{dlg \in Q}) \wedge$
$\overline{m} = (signed\text{-}fpa\text{-}nr,\ Fpa) \wedge (\texttt{Rcvrs}(Fpa) = \texttt{Rcvrs}(Fpa_{st}) + 1)\ ) \vee$
$\exists\ sbj \in \mathcal{CG}, Q_1, Q_2 \subset \mathcal{DG}($**rcvd**$(\langle m, S_{sbj}(m)\rangle)) \wedge |Q_1| = |Q_2| = (2b + d + 1) \wedge$
$m = (fw\text{-}signed\text{-}fpa\text{-}ar, \overline{m}, \{S_{d_1}(\overline{m})\}_{d_1 \in Q_1}, \widetilde{m}, \{S_{d_2}(\widetilde{m})\}_{d_2 \in Q_2}) \wedge$
$\overline{m} = (signed\text{-}fpa\text{-}ar,\ Fpa) \wedge \widetilde{m} = (rec\text{-}merge, IMDD, MReg, Doc, sbj, n_{sbj}) \wedge$
$(\texttt{Rcvrs}(Fpa) = \texttt{Rcvrs}(Fpa_{st}) + 1)) \wedge (N_{\mathcal{IP}}[sbj] = n_{sbj}) \wedge (requests \leq 2)$:
(11.32) $(requests, state, hpm, history) \leftarrow (0, norec, \langle m, S_{sbj}(m)\rangle, history \cup \{\langle m, S_{sbj}(m)\rangle\})$
(11.33) **send**$(\langle m, S_{sbj}(m)\rangle)$ to $\mathcal{DG}$
(11.34) **If** $(m = (fw\text{-}signed\text{-}fpa\text{-}nr, \overline{m}, \{S_{dlg}(\overline{m})\}_{dlg \in Q}))$:
(11.35) $(Fpa_{st}, N_{\mathcal{IP}}[\texttt{Last\_rcv}(Fpa)]) \leftarrow (Fpa, N_{\mathcal{IP}}[\texttt{Last\_rcv}(Fpa)] + 1)$
(11.36) **else**
(11.37) $(Fpa_{st}, Doc_{st}, IMDD_{st}, LSRR, N_{\mathcal{IP}}) \leftarrow \texttt{Stable}(Fpa, Doc, IMDD, N_{\mathcal{IP}}, MReg)$
(11.38) $\overline{m} = (Fpa_{st}, IMDD_{st}, (N_{\mathcal{IP}}[\texttt{Last\_rcv}(Fpa_{st})] - 1))$
(11.39) **send**$(\langle \overline{m}, S_{dl_c}(\overline{m})\rangle)$ to $\texttt{Last\_next}(Fpa_{st})$
(11.40) [] $(state = norec \vee state = rec) \wedge$ **rcvd**$(\langle m, S_{DO}(m)\rangle) \wedge m = (end,$ d_id$)$:
(11.41) **terminate repeat**

Fig. 11. Delegate Protocol

(12.01) $state \leftarrow down$
(12.02) $t \leftarrow$ `Clock`()
(12.03) **repeat**
(12.04) [] $state = down \wedge$ (`Clock`() $- t = MTTIME$):
(12.05) $\overline{\overline{m}} \leftarrow (agreement)$
(12.06) **send**$(\langle \overline{\overline{m}}, S_{dl_c}(\overline{\overline{m}}) \rangle)$ to $\mathcal{DG}$
(12.07) $state \leftarrow$ *sent-agreement-req*
(12.08) [] $state =$ *sent-first-agreement-req* $\wedge \exists\ D \subset \mathcal{DG}(\forall dl \in D$ (**rcvd**$(dl, \langle m, S_{dl}(m) \rangle)$ $\wedge\ m = (agreement\text{-}resp,\ history, hpm, agreements, dlg, n)) \wedge |D| = b + 1 \wedge (n \geq N_{\mathcal{IP}}[dl_c]) \wedge (dlg = dl_c)$:
(12.09) $N_{\mathcal{IP}}[dl_c] \leftarrow N_{\mathcal{IP}}[dl_c] + 1$
(12.10) $Queue \leftarrow$ `Extract-msg`$(\{history\}_{dl \in D}) \cup \{hpm\}_{dl \in D} \cup \{agreements\}_{dl \in D}$
(12.11) $new\text{-}state \leftarrow 0$
(12.12) **repeat**
(12.13) [] $true$:
(12.14) $Queue \leftarrow Queue \cup$ `Received-msg`()
(12.15) $m \leftarrow$ `Higher-priority`$(Queue)$
(12.16) **If** ([(`Type`$(m) \neq$ *fw-signed-fpa-nr*) $\wedge$ (`Type`$(m) \neq$ *fw-signed-fpa-ar*)] $\vee$ $(m = \emptyset)$):
(12.17) **If** $(new\text{-}state = 0)$:
(12.18) **For each** $m \in Queue$:
(12.19) **If** (`Valid`$(hpm) \wedge$ `Priority`$(m) =$ `Priority`(hpm)):
(12.20) $Queue \leftarrow Queue \setminus \{m\}$
(12.21) $Queue \leftarrow Queue \cup \{hpm\}$
(12.22) $m \leftarrow$ `Higher-priority`$(Queue)$
(12.23) `Execute`(m)
(12.24) **terminate repeat**
(12.25) **else** $new\text{-}state \leftarrow 1$
(12.26) `Execute`(m)
(12.27) **terminate repeat**

Fig. 12. Down Delegate Protocol

ALGORITHM 1. *The recovery algorithm (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 recovere d 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*, *MRe g*, *LSRR*)
set-del-docae ← ∅ * *set of elements declared as deleted* *
Sbj ← {`sbj`(Doc)} * *set of subjects from which it has been already received a document version, initialized to the subject that generated doc ument version Doc* *
Rec-Position ← `document-position`(Doc) * *position o f* `sbj`*(Doc) in the Fpa* *
D ← Doc_{st}
(3) * *declarations validity check and last correct document contents search* *
While (*MDD* ≠ ∅):
Doc.//.delete_elmt_cert ← *Doc.//.delete_elmt_cert* ∪ *D.//.delete_elmt_cert*
For each *c* ∈ *Doc.//.delete_elmt_cert* such that (*c.obj.reg*{*r_id*} ∩ *MReg* ≠ ∅):
If ($D_{KU_{DO}}$(*c.signature*) ≠ H(*c* \ *c.signature*) ∨ *c.doc_id* ≠ *doc-id*):
Doc.//.delete_elmt_cert ← *Doc.//.delete_elmt_cert* \ {*c*}
else If (¬∃ $\overline{dd}$ ∈ *MDD.Del-Elmt*: (*c.obj.root_id* = $\overline{dd}$.*doc-root_id*) ∧ (*c.sbj_pk* = $\overline{dd}$.*p_key*)):
Doc.//.delete_elmt_cert ← *Doc.//.delete_elmt_cert* \ {*c*}
For each *c* ∈ *delete_elmt_cert* such that (*c.obj.reg*{*r_id*} ∩ *MReg* ≠ ∅):
If (∃$\bar{c}$ ∈ *delete_elmt_cert* : $\bar{itc}$ ≠ *c* ∧ ∀*r* ∈ *c.obj.reg*{*r_id*} :
c.obj.reg[*r*].*set-docae* ⊆ $\bar{c}$.*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.root_id* = *del-decl.doc-root_id*) ∨
(∃*c* ∈ *Doc.//.delete_elmt_cert*, $\overline{dd}$ ∈ *MDD.Del-Elmt*: *c.obj.root_id* = $\overline{dd}$.*doc-root_id* ∧
∃*r* ∈ *c.obj.reg*{*r_id*} : *del-decl.doc-root_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-d ocae* ∪ *c.obj.reg*[*r*].*set-docae*
(5) * *delete_attr declarations evaluation* *
For each *r* ∈ *MDD.Del-Atrr*{*r_id*}:
Let *Del_attr_decl* be the declaration in *MDD.Del-Attr* for the region *r* with the highest position
If (`position`(*Del_attr_decl*) ≤ `position`(*LSRR*[*r*])):
Doc.FDUR[r].delete_attr_cert ← Doc_{st}.*FDUR[r].delete_attr_cert*
set-del-docae ← *set-del-docae* ∪ *Del_attr_decl.set-docae*
MDD.Del-Attr ← *MDD.Del-Attr* \ {*Del_attr_decl*}
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_pk* = *Del_attr_decl.pub_key*) ∧ (*Del_attr_decl.set-docae* ⊆ *Attr-docae(r)*) ∧ (*c.doc_id* = doc-id) ∧ (*c.priv* = delete_attr) ∧ ($D_{KU_{DO}}$(*c.signature*) = H(c \ *signature*))):
Doc.FDUR[r].delete_attr_cert ← {*c*}
set-del-docae ← *set-del-docae* ∪ *Del_attr_decl.set-docae*
For each *del_attr_decl* ∈ *MDD.Del-Attr* containing region *r*:
MDD.Del-Attr ← *MDD.Del-Attr* \ {*del_attr_decl*}

Fig. 13. The recovery algorithm (part 1)

(6) * *update_attr declarations evaluation* *
For each $r \in MDD.Up\text{-}Attr\{r\_id\}$:
Let Up_attr_decl be the declaration in $MDD.Up\text{-}Attr$ for the region r with the highest position
If (position(Up_attr_decl) $\leq$ position(it LSRR[r])):
$Doc.FDUR[r].update\_cert \leftarrow Doc_{st}.FDUR[r].update\_cert$
$MDD.Up\text{-}Attr \leftarrow MDD.Up\text{-}Attr \setminus \{Up\_attr\_decl\}$
Insertion in Doc of the modifiable content of region r contained in Doc_{st}
else chk $\leftarrow$ Correct
For each $docae \in (P.FDUR[r].DAE\_LIST\{docae\_id\} \setminus set\text{-}del\text{-}docae)$:
Compute $h\_dae \leftarrow$ H($D.FDUR$[r].$DAE\_LIST[docae].data$)
If ($h\_dae \neq D.FDUR[r].DAE\_LIST[docae].h\_docae$): chk = ERR
ãb **break**
If (($D_{KU_{Up\_attr\_decl.pub\_key}}(sig\text{-}fdudocae) =$
H(($\sum^{*}_{t \in D.FDUR[r].DAE\_LIST, type(t)=attribute}$ $t.h\_docae$) $*$ $Up\_attr\_decl.fpa\text{-}id$ $*$
$Up\_attr\_decl.ver$ $*$ $Up\_attr\_decl.rs\text{-}id$ $*$ $Up\_attr\_decl.orig$)) $\wedge$ (chk = Correct) $\wedge$
($\exists$ $c \in D.FDUR[r].update\_cert$: ($Up\_attr\_decl.pub\_key = c.sbj\_pk$) $\wedge$
($c.obj = r$) $\wedge$ ($c.doc\_\ id = doc\text{-}id$) $\wedge$ ($c.priv = update\_attr$) $\wedge$
($D_{KU_{DO}}(c.signature)$ = H($c \setminus signature$)))):
$Doc.FDUR[r].update\_cert \leftarrow \{c\}$
Insertion in Doc of the modifiable content of region r contained in D
For each $up\_attr\_decl \in MDD.Up\text{-}Attr$ containing region r:
$MDD.Up\text{-}Attr \leftarrow MDD.Up\text{-}Attr \setminus \{up\_attr\_decl\}$

(7) * *evaluation of invalid declarations and serch of the next subject to be contact* *
For each $decl \in MDD$:
If (position($decl$) = $Rec\text{-}Position$):
$IMDD \leftarrow IMDD \cup \{decl\}$
$MDD \leftarrow MDD \setminus \{decl\}$
Let $decl\_set$ be the set of declarations in MDD with the same hi ghest position
(declarations of the same subject in a position of the Fpa)
while ($decl\_set \neq null$):
If ($\exists$ $d \in decl\_set$: (($d \in MDD.Up\text{-}Attr \vee d \in MDD.Del\text{-}Attr$) $\wedge$
(position(d) > position($LSRR[d.r\_id]$))) $\vee$ ($d \in MDD.Del\text{-}Elmt$)):
If ($d.pub\_key \in Sbj$):
$IMDD \leftarrow IMDD \cup \{decl\_set\}$
$MDD \leftarrow MDD \setminus \{ decl\_set\}$
$decl\_set \leftarrow$ prev($decl\_set$, MDD)
else **send** ($doc\text{-}id$, $doc\text{-}version$, $doc\text{-}orig$, $pack\text{-}req$) to $d.pub\_key$
If (**rcvd**($d.pub\_key$, D) $\vee$ end($Timeout$)):
If (end($Timeout$)):
$IMDD \leftarrow IMDD \cup \{decl\_set\}$
$MDD \leftarrow MDD \setminus \{decl\_set\}$
$decl\_set \leftarrow$ t̂ prev($decl\_set$, MDD)
else $Rec\text{-}Position \leftarrow$ position(d)
$decl\_set \leftarrow null$
else **For each** $de \in decl\_set$:
If ($de \in MDD.Up\text{-}Attr$):
$Doc.FDUR[r].update\_cert \leftarrow Doc_{st}.FDUR[r].update\_cert$
$MDD.Up\text{-}Attr \leftarrow MDD.Up\text{-}Attr \setminus \{de\}$
Insertion in Doc of the modifiable content of region r contained in Doc_{st}
else $Doc.FDUR[r].delete\_attr\_cert \leftarrow Doc_{st}.FDUR[r].delete\_attr\_cert$
$set\text{-}del\text{-}docae \leftarrow set\text{-}del\text{-}docae \cup de.set\text{-}docae$
$MDD.Del\text{-}Attr \leftarrow MDD.Del\text{-}Attr \setminus \{de\}$
$decl\_set \leftarrow$ prev($decl\_set$, ıt MDD)

(8) * *deletion of document atomic element contents declared as deleted* *
For each $docae \in set\text{-}del\text{-}docae$:
Let reg be the region such that: $docae \in Doc.FDUR[reg].DAE\_LIST\{docae\}$
$Doc.FDUR[reg].DAE\_LIST[docae].data \leftarrow null$

(9) * *return of the algorithm result* *
return (Doc, $IMDD$)

Fig. 14. The recovery algorithm (part 2)