Access control based on CRDTs for Collaborative Distributed Applications

Pierre-Antoine Rault, Claudia-Lavinia Ignat, Olivier Perrin

Access Control based on CRDTs for Collaborative Distributed Applications

Pierre-Antoine Rault*, Claudia-Lavinia Ignat†, Olivier Perrin‡
Université de Lorraine, CNRS, Inria
LORIA, F-54000 Nancy, France
Email: *pierre-antoine.rault@inria.fr, †claudia.ignat@inria.fr, ‡olivier.perrin@univ-lorraine.fr

***Abstract*—A key feature for most collaborative applications is their ability to distinguish access rights to shared documents in a dynamic group of collaborators. To achieve high availability and avoid single points of failure, systems can replicate this access control policy across sites. Increasingly, Conflict-free Replicated Data Types (CRDTs) are used to this end. They embed conflict resolution strategies that take into account concurrent modifications to the policy but also to the document that it manages. However, few systems in the literature support multiple administrators, and those which do fall short of considering distributed applications where no node or group of nodes is central to its operation. To allow multiple concurrent edits of the policy in massively collaborative settings with no central server, we devise a specialized causal model that minimizes concurrency. Crucially our model allows to correct the document state in case operations have become unauthorized following a policy change. We apply our model to a CRDT replicating an access control policy with *read* and *write* rights on a collaborative document and *admin* rights on the policy. We end by studying flexible conflict resolution strategies, providing our underlying algorithms.**

***Index Terms*—distributed algorithms, access control, CRDT (Conflict-free Replicated Data Type), eventual consistency, real-time collaborative editing**

I. Introduction

Collaborative applications are a popular family of message exchange systems aimed to edit documents with other users simultaneously such as articles, wiki pages or program source code. They gained recognition and widespread adoption during the Covid pandemic lock-downs, during which their usage peaked. However, this surge in usage also highlighted their limitations when it comes to large-scale collaboration. Their good operation often relied on a handful of centralized services, making applications vulnerable to the failure of one of them. The sudden load on servers is difficult to predict and yet has to be provisioned for. A common strategy to reduce latency is to bring a copy of the data closer to the user in advance, while fault-tolerance can be increased by server redundancy. However, this approach can be costly and often leads to server over-provisioning.

To improve speed, resilience to faults and attacks, administration cost, scalability, service transparency or even privacy of users [1], collaborative applications increasingly turn to decentralized architectures. To further improve performance and resilience, data can also be replicated across all group members, forming a distributed system. They are however still subject to network partitions, which directly impact user experience by disrupting message distribution [2]. Any distributed system is sensitive to network partitions given Brewer's CAP theorem [3] – whereby consistency, availability and partition tolerance are impossible to achieve together in an asynchronous network. As such, a distributed collaborative application trades consistency off for availability.

In a collaborative exchange, two operations editing a shared document can happen concurrently and be in conflict or lead to divergent states. *Conflict-free Replicated Data Type*s (CRDTs) [4] provide a framework to design data structures that can be replicated across multiple computers or replicas. By their design they avoid incompatible concurrent modifications (conflicts) on the shared structure and allow replicas to be updated without coordination and still converge to the same state.

Another frequent need of collaborative applications is the ability to deal with unauthorized access or modification. That can be specified using rights in a security policy, *e.g.* some users can edit a document, some users can only read it, and some users can even modify users' access to said features. In order to preserve high availability guarantees of the system at scale, access rights have to be replicated and colocated at the user's site like the rest of the application. With simultaneous edits of the policy and the document, conflicts are likely to occur – even more so with the group allowed to edit the policy being dynamic. Some CRDTs already provide conflict resolution rules adapted to a policy and managed document. However, they either use a single administrator per document to arbiter conflicts [5] or rely on synchronization mechanisms to avoid conflicts introduced by supporting multiple administrators [6, 7].

We propose a CRDT-based solution based on our previous work [8] defining the constraints of an access control policy managing a collaborative document with multiple, dynamic administrators. Our CRDT evaluates currently authorized operations based on computed intervals of authorization, and is able to generate the appropriate compensations to the document state, so that it enforces the policy even after unauthorized operations have been integrated. To do so, it relies on a custom dependency tracking mechanism that minimizes the scope of the needed compensation. We discuss and propose strategies combining our contributions to deal with concurrent editions of the document and of the access control policy. After motivating our solution in the section II, we give an overview of the assumptions made for our solution

in section III. We then proceed with a detailed description of its foundational mechanisms in section IV. We describe our underlying algorithm in section V and related work in section VI. Finally, we present concluding remarks and some ideas of future work in section VII.

II. Motivation

Most collaborative applications segregate users' access to features, *e.g.* some can edit a document (W), some can only read it (R), some can modify others' access to said features (A). However distributed systems cannot filter operations through an authorization gateway like their centralized counterparts. Instead, each user interacts with its own instance of the application at their site, rendering a replicated policy locally for the same purpose of availability as for the document. This local copy holds both the document and the rules that *allow* or *deny* authorization to operation modifying the document (document operations). This security policy is enforced locally and ensures all executed operations satisfy its rules.

Seeking high availability, a distributed system should adopt a weaker consistency model such as eventual consistency [9]. In this model, sites eventually converge to a shared state even if messages are received misordered. However, when aiming to enforce access control, ensuring eventual consistency for the policy and the document operations separately is not enough. This is outlined in [5] that caughts on to the need to preserve eventual consistency of document operations w.r.t. policy operations. With no consistency between the two, any misordered policy operations on a site can potentially forbid document operations otherwise already integrated. This situation is avoided in [5] by evaluating anew if operations are authorized, and compensating the state of both policy and document to match the newly computed policy. However, even more recent work such as [10] falls short of allowing multiple administrators to concurrently edit the same policy.

Having multiple administrators, policy operations can be received out of order as shown in Fig. 1: on S_2, receiving $S_1{:}1$ then $S_3{:}1$ produces $S_2{:}A,R,W$ because the first operation is still authorized when received. On S_3 however, $S_3{:}1$ first denies the *admin* right of S_1 to edit the policy, rendering $S_1{:}1$ unauthorized and producing $S_2{:}R,W$. Eventual consistency is therefore also needed for policy operations.

In [11] authors tackle this by making each generated operation reference all operations in its context: a site receiving an operation would thus wait for its referenced operations to arrive before trying to integrate it. However, the bigger the context, the larger the number of operations that can be considered concurrent. The work in [7] proposed to resolve conflicts by choosing the change with the most restrictive set of rights among conflicting changes to the policy. For instance in Fig. 1, the set of resulting rights would yield only the most restrictive of both operations: applying the lower bound for merging conflicting updates corresponds to $S_3{:}1$. By favoring the intention that prevents modifications to the document, this strategy increases integrity of the document. This is however

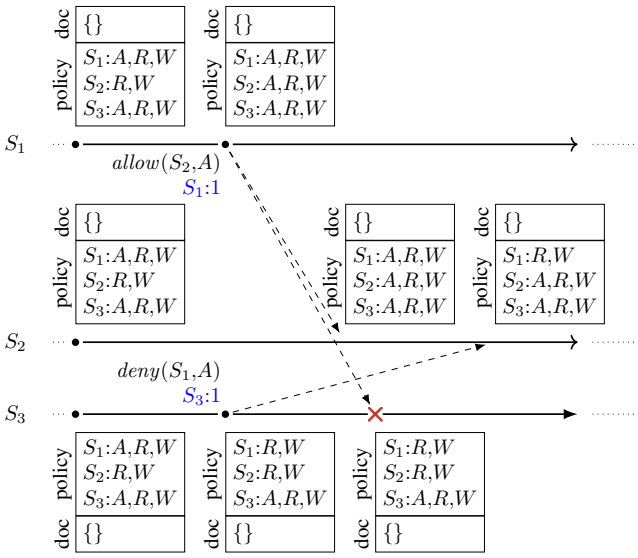


Fig. 1: Divergence of policies after concurrent policy edits

done by reducing accessibility – the rights to read and modify the document – of a site. In [12] authors also provided an extra upper bound strategy favoring accessibility. This stems from their proof of incompatibility between perfect confidentiality (preventing unauthorized viewing), integrity and accessibility in a highly available setting. However, their solution might loose some operations' intentions.

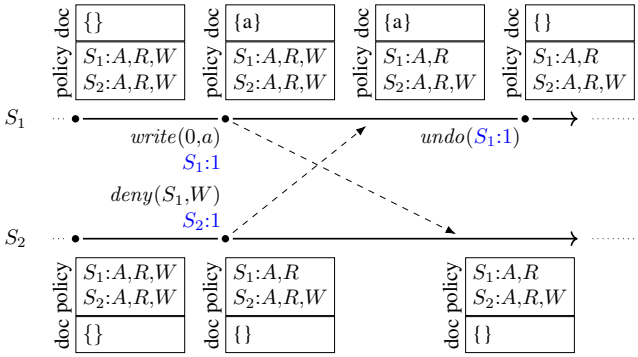


Fig. 2: Convergence of documents after compensation via *undo*

As for conflicts authorizing document operations, the work in [11] favors accessibility and accepts document operations so long as they are made with a local policy which authorized it at the time of their generation. We argue that this limits the applicability of the security policy and contravenes to the integrity (preventing unauthorized modification) of the document w.r.t. the security policy. To strictly enforce the policy would mean to refuse the *write()*, even after $S_1{:}1$ has been applied for at least one site [10]. We refer to this situation as *a posteriori enforcement* of access control, to achieve policy-based consistency. We deem it preferable to using causal consistency frameworks supporting transactions as done in [7] that requires careful provisioning of servers.

To provide *a posteriori enforcement*, we correct the document state by using compensation mechanisms [13, 14, 15]. We apply these compensation mechanisms on the effect of operations that became unauthorized. Operations that are discovered to be unauthorized after a conflict resolution are reverted and operations discovered to be authorized after a conflict resolution are applied again. This is illustrated in Fig. 2 with $undo(S_1{:}1)$: a local correction of the document state to match the currently authorized operations is required to converge. More generally, operations whose effect is already present in the document state could be compensated through an *undo* local operation, and operations not present in the document state could be compensated through a *redo* local operation that just applies their effect.

III. Architecture

We envision in our system a multi-user application where users interact via their own device dubbed *site*. We assume a messaging system where each site executes operations locally, and stores them in a local list of operations called *log*, before replicating them to other sites known as *replicas* as pictured in Fig. 3. Operations are uniquely identified by a site identifier and a number which is incremented for each operation. This tuple *Site identifier*:*operation number* that identifies every operation is called *Dot*. For instance, $S_1{:}1$ is the *Dot* for the first operation of site S_1.

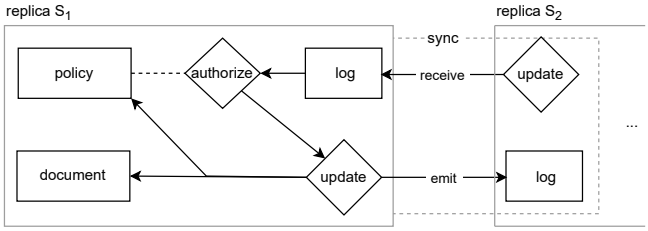


Fig. 3: Components for our application

Operations to be generated are subject to authorization by a replicated security policy. Each replica enforces this access control list where each user has a set of rights granting access to features of our model application. We consider the following features: the right to *read* data (tied to the membership to the group), to *write* data (tied to control of the collaborative document), and the right to change the rights of a user (tied to control of the security policy) as the *admin* right. This is shown in Fig. 3 where a change in document and policy leverages these rights in authorize. The set of rights that can be assigned to a user follow a Role-Based Access Control (RBAC) [16] model, and possible combinations follow common usage in collaborative applications [10]. The group of users managed by the policy can change over time, and so can the subset of users allowed to change the policy. Crucially, an update to the policy might lead to an update to the document: by triggering a re-evaluation of the authorized operations in the `log`, the updates on the document are re-evaluated accordingly; some document updates that were previously canceled might be reactivated. Inversely, a change on the document cannot impact the policy.

A. Delivery model

Sites communicate directly to each other, acting either as client or server for the purpose of message dissemination in what constitutes a *Peer-to-Peer* (P2P) network. Since no site is central to the operation of the distributed application, partitions of the network split sites into disjoint subgroups that can still exchange messages. Messages can be lost, re-ordered or delivered multiple times. To overcome failures of the network, sites rely on a message-passing layer that makes use of an anti-entropy mechanism to synchronize with each other, detect and resend operations lost in the exchange [17].

Each site ensures that the operations are integrated by having their dependencies in their local log beforehand. In the absence of an operation's dependencies in the log, the site stores it into a waiting pool register. Upon reception of new operations, any item of the register whose dependencies are met will proceed with their integration and be removed from the register.

To ensure availability despite network outages and site faults, our system provides eventual consistency via a modelisation of policy state as a CRDT. Our CRDT is composed with another one dedicated to the document being edited. We assume to compose a CRDT for policy with a CRDT for document where document operations have no dependencies with each other and thus are commutative, *e.g.* a Grow-Only Set [4]. A CRDT that needs to keep track of a causal relationship between its own operations such as LogootSplit [18] would subordinate its delivery constraints to our system's.

B. Security and trust model

End-to-End Encryption (E2EE) prevents leakage of information in operation dissemination methods which might rely on intermediaries with no *read* right themselves, such as gossip or relay broadcasting schemes. We rely on distributed group membership protocols that use *Distributed Group Key Agreement* (DGKA) [19]. Our *read* right is thus tied to the right to access and read any operation. Any change to that right would request the underlying DGKA protocol for a key corresponding to the group of sites with that right.

Sending can be conditioned to checking the receiver's *read* right, with senders holding operations as shown in Algorithm CRDT – part 3. It is then trivial to send operations which were undelivered since the last key change once the site gains back its *read* right. Any change to any member's *read* right is also coupled with a pre-requisite group key regeneration with the underlying protocol.

We assume non-administrators to be honest but curious, while administrators are trusted and follow protocol.

IV. Conflict models for access control

A. Causality

Messages exchanged by sites in a system reliant on eventual consistency do not guarantee the same order of reception on each of the sites. A received operation can thus require another not yet present in the *log*. To ensure the dependency from one operation to another, we need a mechanism of

causal consistency by which operations list the operations they require in order to be integrated coherently.

Lamport [20] introduced the commonly accepted definition of potential causality [21] in the context of a message passing system: with each operation having a logical clock C, if operation x potentially causes an operation y then $C(x) < C(y)$. It is said x *happens-before* y, denoted $x \rightarrow y$. This relation is transitive, allowing partial ordering of operations based on their causal relationship. If neither $x \rightarrow y$ nor $y \rightarrow x$, operations x and y are called *concurrent* with each other. The *happens-before* model derives the notion of an operation's *context* [22], which is the sequence of operations received when said operation was generated.

The context of an operation in our application can be limited to a list of the policy operations affecting the same rights, as operations affecting two different rights are independent in our application model. In [23] authors proposed an alternative way of ensuring causal consistency, dubbed *semantic* context, listing only the dependencies semantically relevant to the evaluation of an operation. By applying [23] to our application, semantically unrelated updates to the document and policy under this model would be disconnected, which would allow for more concurrency: it would potentially reduce the range of operations to be undone/redone, in case of re-evaluation of an operation's authorization status. In what follows we provide the necessary definitions for an operation's semantic context.

Definition (access tuple)**.** subset of operations that target the same site and right. Denoted $site\,\{right\}$.

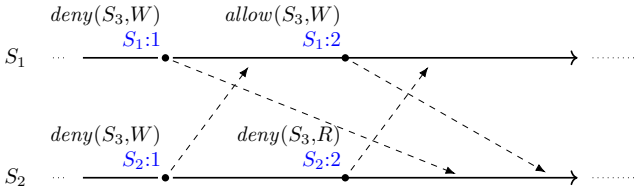
Definition (context [22])**.** list of operations that happened-before an operation.

Definition (semantic context)**.** sublist of the context of an operation, in which every operation is a policy operation and is of same *access tuple*.

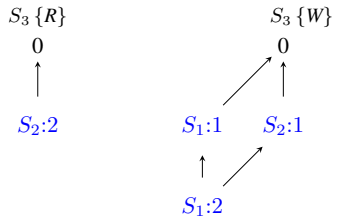
Definition (semantic happens-before)**.** relationship of an operation a to another b, for which a is part of b's semantic context. Denoted $a \xrightarrow{s} b$.

This model of an operation's context can be represented along directed acyclic graphs, in which the root of the graph (or initial state) is then used as dependency by other operations. Regardless of the order of their arrival on each site in Fig. 4a, operations can thus be represented via the same causal representation in Fig. 4b, grouped by access tuple: $S_3\,\{W\}$ (*write* right for the site S_3) for $S_1{:}1$ and $S_2{:}1$; $S_3\,\{R\}$ (*read* right for the site S_3) for $S_2{:}2$. Both access tuples are independent: operations in one will not affect those in the other.

This way, policy operations of same access tuple within the *log* can be reasoned about by organizing them along a graph of the policy operations they refer. Adding a new operation $S_1{:}2$ in access tuple $S_3\,\{W\}$ of Fig. 4b can be done by considering only the last known policy operations of that sub-graph in the *log*. $S_1{:}2$ only needs to list $S_1{:}1$ and $S_2{:}1$ as dependencies, as they are the last operations of the access tuple.



(a) Execution timeline for operations



(b) Causal representation for the same operations

Fig. 4: Representation of operations grouped by *access tuple*

Definition (level h)**.** subset of an access tuple, in which every operation is of the same shortest path distance *h* to the initial state. Each level corresponds to a transition from a state of the policy (allowing/denying) to the other. Each level holds operations of *opposite effect* (allowing/denying) to the ones in the previous and next level.

Definition (authorized operation)**.** an operation is authorized if it does not lose any conflict with concurrent operations, and is allowed by the current policy.

The current policy corresponds to the effect of authorized operations in the level of greatest h. Therefore a generated operation only lists the last authorized policy operations of its semantic context, and can be thought as creating a *level* in its access tuple. The semantic context of a newly generated operation is always the latest level with authorized operations furthest of the root. Thus in Fig. 4b the operation $S_1{:}2$ is of level 2.

Definition (parents/dependencies)**.** subset s of the semantic context of an operation o, where s corresponds to the greatest level of authorized operations when o is generated. o should wait for and be applied after every operation in s. By extension, *ancestors* refer to the recursive lookup of parents until the access tuple root.

We need to establish a partial order in which to apply operations. For instance in Fig. 4b, $S_1{:}1$ and $S_2{:}1$ are concurrent *because* they are not dependent on one another but on the initial state "0". Through metadata added to the structure of operations as presented in Basic Type 1, we should be able to inform if an operation is *concurrent with* or *dependent on* another operation, by checking their `Deps` attribute, i.e. the list of its dependent operations.

Basic Type 1

```
type Dot : ⟨Site identifier, operation number⟩
type BasicOperation : ⟨
    Type := policy
    Dot_Source : Dot
    ID_Target : Site identifier
    Right : admin | write | read
    Effect : allow | deny
    Deps : Dot[]
  ⟩ or ⟨
    Type := document
    Dot_Source : Dot
    Right := write
    Deps : Dot[]
  ⟩
```

B. Basis for conflict

Generated locally, a policy operation or authorized document operation is integrated immediately. Received from a remote site however, an operation is integrated only after all its dependencies have been authorized and integrated. Evaluating if a policy operation is authorized requires to check not only the emitter right to modify the policy, but the operation parents' authorization too, as in Algorithm 1. This evaluation begets conflict detection for policy operations. Two policy operations that are concurrent, of different *level*, and of opposite effect are said to be in conflict. A policy operation is deemed authorized once all conflicts have been cleared in its favor.

Algorithm 1 validity evaluation for *policy* operations

```
1: isValid(o : policy operation) : Boolean
2:     concurrents := set of concurrent operations among ancestors of o
3:     conflicts := set of operations of opposite effect in concurrents
4:     if conflicts is ∅ or
         no valid conflict lost by o with any operation in conflicts then
5:       if dependencies of o are authorized then valid := True
6:       else valid := False
7:     else
8:       valid := False
9:     if valid changed then inform onValidityChange (see page 7) for o
10:    ret valid
```

A consequence of making any policy operation prevail on document operations is that policy operations need to reference the document operations of which they had knowledge. Without that metadata, all document operations referencing the same parent policy (`Deps` in Basic Type 1) would be concurrent with any new policy operation. To prevent any document operation from becoming invalid when a competing policy operation exists, we need policy operations to keep track of those document operations seen at the generation time.

C. Conflicts between policy and document operations

Policy operations prevail when in conflict with document operations. Suppose two operations – one policy, the other document-related where their dependencies are in the *log* at the time of their integration. Further suppose these two operations share the same dependencies and are concurrent. In this case the policy operation will prevail. But if the policy was generated on a site where the document operation *document semantic happens-before*, the latter would not be invalidated.

Definition (document semantic happens-before)**.** relationship of an operation a to another b, for which a is part of b's document semantic context: a happened-before b, a is of same access tuple as b, and a is a document operation. Denoted $a \xrightarrow{\text{ds}} b$.

The policy operation must thus be accompanied by the Dot of the last document operation managed by their access tuple at the time of their generation since the last policy generation, denoted `LastDotSeen`. Policy operations are also sent with a list of Dots corresponding to operations that have to be invalidated, denoted `MissingDots`. `LastDotSeen` and `MissingDots` allow another site receiving both operations to check which document operation within the policy *access tuple happens-before* the policy change. We extend the BasicOperation in Basic Type 2 with `LastDotSeen` and `MissingDots` for policy operations.

Basic Type 2 $\rightarrow$ extension of BasicOperation

```
type Operation : BasicOperation(Type := policy) & ⟨
    LastDotSeen : Dot
    MissingDots : Dot[]
  ⟩
```

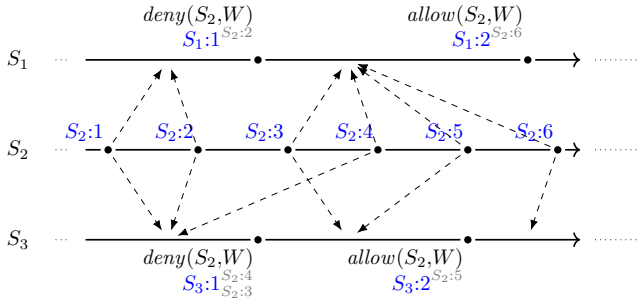
We also extend the notation for policy operations with the last known document operation as superscript and eventual excluded list thereof as subscript. For instance in Fig. 5b, $S_2{:}1 \xrightarrow{\text{ds}} S_1{:}1$ and $S_2{:}2 \xrightarrow{\text{ds}} S_1{:}1$, so we can denote $S_1{:}1^{S_2:2}$ to explicit that relation. Additionally the excluded Dots list can be made explicit as in Fig. 5c where $S_2{:}3$ is excluded by $S_3{:}1$, denoted $S_3{:}1^{S_2:4}_{S_2:3}$.

When a document operation is received at a site, we need to check whether concurrent policy operations invalidate it. To do so, we take into account all the lists of a *level*, and make a synthesis of them, in the form of authorization intervals.

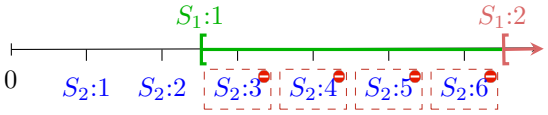
Definition (authorization intervals)**.** For each access tuple, we have a set of policy operations changing the right. Each policy operation holds information regarding which document operation it has knowledge of. This maps document operations to intervals in which the right is enabled or disabled. Any document operation thus can be mapped between two levels of their access tuple.

Let us take the example of document operation $S_2{:}6$ issued by S_2 in tuple $S_2\{W\}$ in Fig. 5a : concurrently, a policy operation $S_1{:}1^{S_2:2}$ issued by S_1 denies to S_2 the right to modify the document. Its list of dots then shows that S_1 is aware of the operations issued by S_2 up to $S_2{:}2$, thus not of $S_2{:}6$. Any document operation of number higher than 2 should therefore be invalidated. S_1 adds next an additional policy operation $S_1{:}2^{S_2:6}$ allowing again S_2 to modify the document. Even though $S_2{:}6$ is in the dependencies of $S_1{:}2$ at the time of the opening of the rights, $S_2{:}6$ stays invalidated.

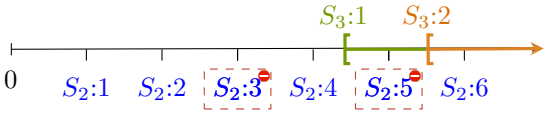
If some document operations were received in misorder, it is possible to exclude the non-received operations at the



(a) Execution timeline for operations of $S_2\{W\}$



(b) Validity intervals of $S_2\{W\}$ computed by S_1



(c) Validity intervals of $S_2\{W\}$ computed by S_3

Fig. 5: Validity intervals for $S_2\{W\}$ before S_1 and S_3 sync – omitting site state as irrelevant

policy generation by means of the `MissingDots` field, *e.g.* $S_3{:}1^{S_2:4}_{S_2:3}$ informs that S_3 did not see $S_2{:}3$ when issuing $S_3{:}1$, thus rendering operation $S_2{:}3$ invalid. Following the misorder, the two concurrent intervals in Figs. 5b and 5c need to be synchronized and still converge. For instance S_3 receives $S_1{:}1$ whose metadata conflicts with $S_3{:}1$. It should still compute a single interval: to do so, we can either keep only a single operation' metadata, keep part or keep all lists. We show how to merge these metadata using `LastDotSeen` as boundary below, and show how the interval is computed in Algorithm 2.

Algorithm 2 validity evaluation for *document and policy* operations

```
1: isValid(o : operation) : Boolean
2:     ops := level of o and newer levels
3:     intervals := list of boundaries from ops
4:     excluded := set of excluded operations from ops
5:     if o not in excluded and within authorization of intervals then
6:         valid := True if dependencies of o are valid
7:         if valid changed then inform onValidityChange (see p.7) for o
8:     ret valid
```

Evaluating whether the operation is within authorization of *intervals* (line 5) is done by merging the operation lists for a path in the access tuple DAG. If more than one path exist, we can merge interval lists by electing the upper or lower bounds, depending on the strategy. Using an upper bound will postpone the enforcement of the change, whereas a lower bound will hasten it.

Merging intervals via a lower bound: We define the lower bound of a level as a relation min on type Dot, which returns the Dot with operation number n such that $n \leq n' \ \forall n'$. Applying the min relation to a level, we get the Dot of the operation with the lowest operation number within those listed as `LastDotSeen` in that level.

Merging intervals via an upper bound: We define the lower bound of a level as a relation max on type Dot, which returns the Dot with operation number n such that $n \geq n' \ \forall n'$. Applying the max relation to a level, we get the Dot of the operation with the highest operation number within those listed as `LastDotSeen` in that level.

Combining a lower bound for denying a right and a upper bound for allowing a right corresponds to a strategy favoring *integrity* of the document, with the case $n = n'$ denied. The inverse strategy favors *accessibility* to edit the document, with $n = n'$ allowed.

D. Conflicts between policy operations

Two policy operations can be in conflict if they are of same *access tuple* but opposite effect (one *allows* and the other *denies* the right). Only operations from different levels are thus considered for conflict. Winning a conflict indicates the operation's effect will prevail. To favor one operation over another, we can choose among arbitrary differentiating orders (*e.g.* lexicographic order of senders ; non-monotonic clock [24], *etc.*), or systematically favor operations allowing/denying rights – favoring availability/confidentiality *resp.*

It is possible for operations of different access tuples to be in conflict, as concurrent operations can deny each other the *admin* right. This cycle can be defeated by totally ordering administrator operations in that case, following the same differentiating order applied to administrators. However, to properly decide if another modification of the policy is concurrent to them, policy modifications affecting an *admin* right access tuple should bear extra metadata: the dependencies should be extended to include other admins' last operation seen.

V. ACCURE: An Access Control CRDT with Undo/Redo

The replicated data structure ACCURE[1] on each site maintains the information necessary to evaluate the access policy and thus the document state at all times. To this end, each site populates its own *log*, a List to which newly integrated operations are appended upon reception. Operations are identified by their *Dot*s, whose numbering is contiguous within the same access tuple. Operations bear a *type* identifying them as policy or document operations, with according attributes. Among those are `Deps` tracking semantic causality, and the Dots `LastDotSeen` and `MissingDots` serving to evaluate document operation validity.

Each operation allowing/denying a right is done through an update of the policy: it adds a node to the G Map$\langle access\ tuple \rightarrow dag \rangle$ register, which to each *access tuple* associates a Add-only Monotonic DAGs in Algorithm CRDT – part 1. The value of our policy lies in the interpretation of data stored in `G`. `G` holds operations layered

[1] ACCURE test code available at gitlab.inria.fr/coast-team/crdt-ac/tests

in levels according to their dependencies, and is the one part of the CRDT that needs convergence with other sites.

CRDT – part 1 – update

```
1:  state σ :
2:      local log : List⟨Operation⟩ := ∅
3:      replicated G : Map⟨⟨Site identifier, Right⟩ → Add-only Monotonic
        DAG⟨Dot[], Dot⟩⟩ := initial policy
4:      local Undelivered : Map⟨Site identifier → Set⟨Operation⟩⟩

5:  update policy (i : Site identifier, d : Right) : Operation
6:      pre eval(Current site identifier, admin) = True
7:      Dot_Source := Current site identifier + Autoincremented
8:                    operation number per emitter
9:      LastDotSeen := last known Dot for access tuple ⟨i,d⟩
10:     MissingDots := list of missing dots deduced from the
11:                    last known Dot and σ.log
12:     Deps := list of valid policy operations of same level as
13:             operation o of longest arc of σ.G[⟨i,d⟩]
14:             such that isValid(o)
15:     Effect := deny if eval(i, d) = True, else allow
16:     ret Operation⟨policy, Dot_Source, i, d, Effect,
        LastDotSeen, MissingDots, Deps⟩

17: update document (): Operation
18:     pre eval(Current site identifier, write) = True
19:     Dot_Source := Current site identifier + Autoincremented
20:                   operation number per emitter
21:     Deps := list of valid policy operations of same level as
22:             operation o of longest arc of σ.G[⟨ Current site
23:             identifier, write⟩] such that isValid(o)
24:     ret Operation⟨document, Dot_Source, Deps⟩
```

Each update is subject to a precondition checking whether the relevant access tuple authorizes the modification: the update document function checks the local state of the policy for a *write* right (line 18) ; the update policy function is conditioned to an analogous precondition for the *admin* right (line 6).

A locally generated operation is integrated via effect both locally (adding the operation data to the *log* and to G[tuple], lines 27–29) and on remote sites after being sent over the wire by triggering a sending event.

A received operation, in addition to adding the operation data to G, uses validity tracking to help manage the state of the document (lines 30–35). This state of validity is stored and evaluated first at the document effect (lines 38–39).

CRDT – part 2 – effect integration

```
25: effect(o : Operation | o.Type = policy)
26:     tuple := σ.G[⟨o.ID_Target, o.Right⟩]
27:     for all Dep ∈ o.Deps do
28:         tuple.AddEdge(Dep,o.Dot_Source)
29:     add o in σ.log
30:     updateValidityTracking(o, isValid(o))
31:     for all t ∈ σ.LastValidity.keys | t.Type = policy do
32:         tuple := σ.G[⟨t.Dot_Source,o.Right⟩]
33:         for all c ∈ tuple do
34:             if level(c) = level(t) ∧ c.Type = document then
35:                 updateValidityTracking(c, isValid(c))

36: effect(o : Operation | o.Type = document)
37:     add o in σ.log
38:     if isValid(o) = True ∧
          eval(Site identifier of o.Dot_Source, o.Right) = True then
39:       apply the effect of o to the document
```

We manage the state of the document by evaluating the validity of document operations and their transition from one valid/invalid state to another when a policy operation is added. It allows to decide which operation will need compensation by generating its undo/redo, respectively. The compensation is done by generating, for each operation to be compensated, a local operation that is not broadcast.

CRDT – part 3 – validity tracking

```
40: on validityChange(o: Operation, newValidity: Boolean)
41:     if o.Right = read ∧ (
            (o.Effect = allow ∧ newValidity = True) ∨
            (o.Effect = deny ∧ newValidity = False)
        ) ∧ eval(o.ID_Target, read) = True                    then
42:       key = regenerate group_encryption_key
43:       for all o′ ∈ σ.Undelivered[o.ID_Target] do
44:           send enc(o′, key) to o.ID_Target
45:           remove o′ from σ.Undelivered[o.ID_Target]

46: eval(i : Site identifier, d : Right) : Boolean
47:     o := last operation of the longest arc within σ.G[⟨i,d⟩] such that
            isValid(o)
48:     ret True if o.Effect = allow else False
```

First we initialize the value for o in a register storing validity (line 30), then we record changes to that evaluation for operations whose validity was impacted by the addition of o. Thus we hold all the information necessary to apply corrective operations to the document's local state (also necessary to avoid double edits of the document through over-compensation), or to react to the allowance of a *read* right, by sending pending messages stored in Undelivered for this member (lines 43–45). The value of the policy can be evaluated by running eval (lines 46–48).

VI. Related Work

Maintaining dynamic security policies under high availability constraints is a problem known across different areas of research. Early on with database access control, replicated access control was suggested [25]. For maintaining authorization consistency, concurrency control techniques generally use explicit locking or transaction processing which are inappropriate for replicated environments. Bouganim et al. [26] proposed a client-based evaluation of access control rules for regulating access to documents based on the access control model of [27]. However, this evaluation only concerns users that can read but not update shared documents. Our approach targets scenarios where users update shared documents concurrently with changes on access control policies.

Write authorizations in weakly consistent state-based replicated systems can be implemented in the policy management by designating a trusted member for all collaborating items [5] or a fixed administrator per document [10]. The crash of this single root would create security issues and block the collaboration unless they delegate all their capabilities to another user. Our approach deals with several administrators.

A model closer to our requirements revises the semantics of access control and provides a model of dynamic administrators [11]. Through highly available transactions guarded by an access control policy and CRDT, its causal model allows a

grace period of synchronization between sites before enforcing policy changes. It avoids policy-document conflicts thanks to Cure's latency upper bound computation [6]. Our approach is fully distributed and cannot deploy Cure on dynamic administrator sites.

Confidentiality, integrity and accessibility cannot all be guaranteed at the same time by any system. Alternative merging semantics for CRDTs that focus on information accessibility – but not on integrity of the document – have been defined [12]. Our approach deals with potentially massive collaborative applications, where non-administrator sites cannot be trusted at scale, and whose edits thus cannot be kept in case of concurrency with a policy update denying the right to edit.

VII. Conclusion

Addressing the need for a CRDT enabling access control for collaborative applications, this paper provided novel mechanisms for multiple dynamic administrators and no reliance on server-dependent causality mechanisms. By defining a causality tracking mechanism around semantic causality, we also minimized concurrency for our rights representation. Addressing the challenge of stronger policy enforcement and document integrity under multiple administrators, we provided a novel compensation mechanism that allows corrections to the document state to match the currently authorized operations.

Our implementation proved the feasibility of our model under certain conditions. Possible avenues of improvement encompass adaptation for dependent rights, providing a full suite of convergence tests, taking into account other conflicts arising from its effective use with E2EE, and integration within the real-time distributed collaborative editor MUTE [28].

References

[1] R. Rodrigues and P. Druschel, "Peer-to-peer systems," *Communications of the ACM*, vol. 53, no. 10, pp. 72–82, Oct. 2010.

[2] A. Tseitlin, "The antifragile organization," *Communications of the ACM*, vol. 56, no. 8, pp. 40–44, Aug. 2013.

[3] A. Fox and E. Brewer, "Harvest, yield, and scalable tolerant systems," in *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, Mar. 1999, pp. 174–178.

[4] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-Free Replicated Data Types," in *Stabilization, Safety, and Security of Distributed Systems*, vol. 6976. Springer Berlin Heidelberg, Jan. 2011, pp. 386–400.

[5] T. Wobber, T. L. Rodeheffer, and D. B. Terry, "Policy-based access control for weakly consistent replication," in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10. Association for Computing Machinery, Apr. 2010, pp. 293–306.

[6] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro, "Cure: Strong Semantics Meets High Availability and Low Latency," in *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, Aug. 2016, pp. 405–414.

[7] M. Weber and A. Bieniusa, "ACGreGate: A Framework for Practical Access Control for Applications using Weakly Consistent Databases," *arXiv:1801.07005 [cs]*, Jun. 2018.

[8] P.-A. Rault, C.-L. Ignat, and O. Perrin, "Distributed Access Control for Collaborative Applications using CRDTs," in *PaPoC 2022 - 9th Workshop on Principles and Practice of Consistency for Distributed Data*, Apr. 2022, p. 7.

[9] W. Vogels, "Eventually Consistent: Building reliable distributed systems at a worldwide scale demands trade-offs between consistency and availability." *Queue*, vol. 6, no. 6, pp. 14–19, Oct. 2008.

[10] A. Cherif, A. Imine, and M. Rusinowitch, "Practical access control management for distributed collaborative editors," *Pervasive and Mobile Computing*, vol. 15, pp. 62–86, Dec. 2014.

[11] M. Weber, A. Bieniusa, and A. Poetzsch-Heffter, "Access Control for Weakly Consistent Replicated Information Systems," in *Proceedings of International Workshop on Security and Trust Management*, ser. STM 2016. Springer International Publishing, Sep. 2016, pp. 82–97.

[12] E. Yanakieva, M. Youssef, A. H. Rezae, and A. Bieniusa, "On the Impossibility of Confidentiality, Integrity and Accessibility in Highly-Available File Systems," in *Proceedings of the International Conference on Networked Systems*, ser. NETYS 2021. Springer International Publishing, Dec. 2021, pp. 3–18.

[13] Y. Mao, "Reversing CRDTs Through Compensating Operations," Thesis, Université de Toronto, Nov. 2021.

[14] W. Yu, V. Elvinger, and C.-L. Ignat, "A Generic Undo Support for State-Based CRDTs," in *Proceedings of the 23rd International Conference on Principles of Distributed Systems*, ser. OPODIS 2019, Dec. 2019.

[15] W. Yu and C.-L. Ignat, "Conflict-free replicated relations for multi-synchronous database management at edge," in *The IEEE International Conference on Smart Data Services (SMDS 2020)*, Oct. 2020, pp. 113–121.

[16] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-based access control models," *Computer*, vol. 29, no. 2, pp. 38–47, Feb. 1996.

[17] D. Parker, G. Popek, G. Rudisin, A. Stoughton, B. Walker, E. Walton, J. Chow, D. Edwards, S. Kiser, and C. Kline, "Detection of Mutual Inconsistency in Distributed Systems," *IEEE Transactions on Software Engineering*, vol. SE-9, no. 3, pp. 240–247, May 1983.

[18] L. André, S. Martin, G. Oster, and Claudia-Lavinia Ignat, "Supporting adaptable granularity of changes for massive-scale collaborative editing," in *Proceedings of the 9th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing*, ser. CollaborateCom 2013. ICST, Nov. 2013, pp. 50–59.

[19] M. Weidner, M. Kleppmann, D. Hugenroth, and A. R. Beresford, "Key Agreement for Decentralized Secure Group Messaging with Strong Security Guarantees," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. Association for Computing Machinery, Nov. 2021, pp. 2024–2045.

[20] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[21] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto, "Causal memory: Definitions, implementation, and programming," *Distributed Computing*, vol. 9, no. 1, pp. 37–49, Mar. 1995.

[22] S. Burckhardt, "Principles of Eventual Consistency," *Foundations and Trends in Programming Languages*, vol. 1, no. 1-2, pp. 1–150, Oct. 2014.

[23] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "The potential dangers of causal consistency and an explicit solution," in *Proceedings of the Third ACM Symposium on Cloud Computing*, ser. SoCC '12. Association for Computing Machinery, Oct. 2012, pp. 1–7.

[24] S. S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone, "Logical Physical Clocks," in *Principles of Distributed Systems*, ser. Lecture Notes in Computer Science. Springer International Publishing, Dec. 2014, pp. 17–32.

[25] P. Samarati, P. Ammann, and S. Jajodia, "Maintaining replicated authorizations in distributed database systems," *Data & Knowledge Engineering*, vol. 18, no. 1, pp. 55–84, Feb. 1996.

[26] L. Bouganim, F. D. Ngoc, and P. Pucheral, "Client-based access control management for XML documents," in *Proceedings of the 30th International Conference on Very Large Data Bases - Volume 30*, ser. VLDB '04, Aug. 2004, pp. 84–95.

[27] P. Samarati and S. C. de Vimercati, "Access Control: Policies, Models, and Mechanisms," in *Foundations of Security Analysis and Design*, ser. Lecture Notes in Computer Science. Springer, Oct. 2001, pp. 137–196.

[28] M. Nicolas, V. Elvinger, G. Oster, C.-L. Ignat, and F. Charoy, "MUTE: A peer-to-peer web-based real-time collaborative editor," in *Proceedings of the 15th European Conference on Computer-Supported Cooperative Work*, ser. ECSCW 2017, vol. 1. EUSSET, Aug. 2017, pp. 1–4.