Dynamic Access Control for Transactional Causal Consistent System

Romain du Marais, Marc Shapiro, Mathias Weber, Annette Bieniusa
Laboratoire d'Informatique de Paris 6, REGAL team, 2017

1. Introduction

With the growing number of highly available applications, the success of geo-replicated databases is increasing, raising serious consistency issues. One of these issues is to provide security guarantees in the access of the replicated data. To prevent such issues, specific access control systems are designed, but consistency issues in access control systems may lead to data leakage or tampering. In current access control models, a problem is to deal with both application-level operations and access to database-level resources, and still ensure consistent and modular access management.

There are two approaches to solve this issue, used respectively by ACGreGate and S3 Access Control. ACGreGate has been deisgned for geo-replicated databases, but is neither adaptative to changes in the Access Control, nor as general as desired. S3 Access Control Management uses redundancy in the access resources to provide the most restrictive Access Control as possible. But it is actually not fitted for the desired consistency models.

Our aim is to design a model of Access Control that is both as restrictive and modular as S3 and as consistent as ACGreGate. My approach will be then to build over the ACGreGate model the S3 features. This approach is quite heavy to deal with but ensures the TCC guarantees with the features of S3.

In the Section 2 of the current report, we discuss the security issues in geo-replicated systems. In Section 3 we will see how an Access Control issues may solve these issues if it ensures some conditions. Section 4 will explain the need and the approach of ACGreGate model for Access Control and its implementation. We will discuss how it solves somes Access Control problems, but still raises issues, that need to be filled in a different model. In Section 5 and 6, we will detail the DAcCoRD model proposal and implementation, before concluding and explain remaining work for securing Antidote in Section 7.

2. Security in geo-replicated systems

2.1 Cloud Databases

The majority of modern applications store their data in several Data Centres (Dcs) to provide high availability and low latency. The data is then replicated between the different DCs, called **replicas**. However, application developpers may face therefore unpredictable behaviors brought by the difficulty to synchronize this replicated data. Providing consistency guarantees for these geo-replicated data storage is now a major concern.

A **cloud database** can be defined as a database where the data storage is shared between the data centers and can be accessed via Internet.

2.2 Antidote database

Antidote Database has been created for the Syncfree european research project[1]. It is a database for geo-replicated and scalable data storage. Antidote offers rather important consistency guarantees and high availability using the Transactional Causal Consistency model[2]. Antidote is based on operation-based CRDTs.

Antidote is currently not a file system, but a key storage system using buckets for object management[3].

2.1.1 Object types

The main types of CRDTs are the following[4] :
- Counter : basic container for an integer, that supports *set*, *increment* and *decrement* operations.
- Register : a unique field, that can be anything. The only operation supported is the assignment of a new value. Conflict resolution may use *Last-Writer-Wins* or *Multi-Value* (every concurrent update is stored and the set of concurrent values is returned) semantics.
- Set : a list with *add* and *remove* updates. The level of preference between both operations can be set regarding the type of set used.
- Maps : map binding a key with a CRDT object. It forwards all operations to the embedded CRDTs.

2.1.2 Lifecycle

Currently in Antidote, object and buckets do not have any well-defined life cycle. There is no such operations as « creating an object/bucket » or « deleting the object/bucket ». If a transaction updates a CRDT for the first time, it will initialize it. But if the first transaction to use a key is a « read » operation, the read will actually return a default value.

2.3 Security issues in cloud databases

Restricting access to distributed databases is an important issue to prevent data leakage. In the case of cloud databases, everything may be accessed from an internet interface. Therefore it is essential to prevent any data to be publicly disclosed. It is moreover important to separate access rights with a fine granularity. In the case of a database shared between two companies, a bad-designed database could enable employees from one company to read and even tamper personal information on the other company’s employees.

2.4 Security issues in weakly consistent databases

In weakly consistent databases, the issue of data leak and tampering is complex to handle. Indeed, if on a replica 1, a subject (an actor perfoming operations on the database, e.g a user) *A* can access a data, for example personal information on the company’s employees, and a subject *B* decides concurrently to revoke the access of user *A*, for example he quit the company. If the subject *B* is connected to another replica 2, and in the case of a partition, in a weakly consistent context, the subject *A* may still read or modify the personal informations he’s not supposed to. Depending of the consistency model, the subject *A* may even see an arbitrary number of updates of this data before the access revocation is effective[5][6].

Another issue to deal with is in the contrary complex to provide, is that it will be always possible for someone to access an object *object*. For example, if two subjects *A* and *B* have full control on an object *obj*. If A makes an update on a replica 1 to give the subject *B* the access of the object *obj*, and concurrently the subject *B* transfers its control to *A* in the same way on the replica 2. The merging of the two updates may lead to a situation where none of the two subjects can access the data *obj*.

3. Access Control

Managing Access Control for replicated systems tend to solve the access issue for a geo-replicated databases.

3.1 Access Control definition

An Access Control System restricts access to data to ensure :
- that unauthorized users can not access some data ;
- that only authorized users can use the systems ;
- that a user can access a data only if this user is allowed to.

The Access Control is the set of two items : an Access Monitor, that intercepts the requests to the database, and the Decision Procedure, that provides Access Control Decisions. A **decision** is a function that returns a boolean given an access **request** and a set of **access resources**. A request a set with an operation, a subject, an object and a context. The **operation** is the operation to perform remotely in the database : either a *read* or an *update*. The **subject** is the user that is currently performing the operation. The **object** is the set of data on which the operation is performed. The **context** is the application-layer properties of the environment performing the request, for example the location or the date of the request.
The access resources are the set of objects defining the **permissions**, i.e the set of statements used as rules to make an access decision.

3.1.1 Abstraction

An access decision can be formalized as follows :

$$dec : Requests \times \wp(P) \rightarrow \{ ALLOW , DENY \}$$

$$dec : (request , pol) \mapsto x$$

with :

- *dec* : the access decision

- *request* : The request is defined as : $request = \{ op , subj , obj , contx \} \in Requests$, with $Requests = (Ops \times Subjs \times Resrc \times Env)$. We note :

  - *subj* : the subject of the operation, which may i.e the user ID. We note *Subjs* the set of subjects.

  - *obj* :the object on which the operation is performed, that may be an data, a file, a bucket, a directory, or a set of these resources. We note *O* the set of objects in the database, and *Resrc* the set of data in the database.

  - *op* : the currently requested operation. We note *Ops* the set of possible operations.

  - *contx* : the context of the request, it is a piece of information on an event in the set of database calls. The set of context objects is noted *Env.*

- *pol :* the set of access control resources needed to compute the access decision. We note $P \subset Rsrc$ the set of access resources in the database, and $\wp(P)$ the set of the subsets of P.

- *x* : a boolean, set to *ALLOW* for an authorized access or *DENY* for a denied access

3.1.2 Access Control Properties

An Access Control verifies four properties, summed up by the acronym « NEAT », which stands for : « Non-bypassable, Evaluable, Always invoked, Tamper-proof »[7] .

- Non-bypassable : An user can not perform an access if the corresponding access decision result was negative.
- Evaluable : Every properties of the Access Control System can be tested.
- Always invoked : It is not possible to access the protected data without requesting an access decision to the Access Control System.
- Tamper-proof, i.e non undermineable by an attacker : An unauthorized user can not modify the Access Control, nor the Access Control Lists involved in the process that makes the access decisions.

3.2 Access Control issues

The main goal of Access Control is to provide guarantees against data leakage and tampering. However, to solve this issue via an Access Control raises 3 main issues.

3.2.1 Access Control gap Issue

In practice there are two different approach for an Access Control :

- Operation-based Access Control
  In this case, the Access Control allows or denies a request based on the application-layer operation that is requested. For example, in a shopping application, a subject has the is allowed to buy what is in the shopping cart under some conditions.
  This type of access control is based on the requested operation *op (*which may be any of the set *A* of operations), and the context *contx*. However, the requested object does not describe the whole set O, since the application-layer operation is not bound to specific resources, i.e the function $request \rightarrow \{op, obj\}$ may be surjective in Ops but can not be surjective in *Rsrc*.

- Ressource-based Access Control
  In this case, the access decision is made based on the database-layer objects that are requested. For example, in a file storage application, a user has to read or write in a file or not.
  This type of access control is based on the object *obj*, that describes the complete set of objects *O*. Tshe function $request \rightarrow \{op, obj\}$ may here be surjective in $O \subset Rsrc$. But the set of operations used is restricted to a very small subset of *Ops*, for example :
  $\{read, write, writePermissions, readPermissions, changeOwner\} \subset Ops$.

Resource-based Access control is usually implemented at a database layer, which makes it easier to ensure the *Always-Invoked* property. Operation-based Access Control is usually implemented an Application Layer level. It prevents the client to request arbitrary or tampered operations, which makes it easier to provide the *Tamper-Proof* property. There are two ways to conciliate the two approaches [8] :
- design an access control at the application layer and intercept the calls to the database and perform both types of access control.
- design an access control at the database level and send to the database information about the request, e.g the operation to perform and the context.

3.2.2 Geo-replicated context : permission delegation issue

An issue for Access Control in a geo-replicated context is the replication of the permissions management. As exlained before, if an update modifies the access permission of an user, the databases needs to ensure :
- either that every policy it uses in an access decision is the latest version available
- either that a requested operation *Op* is performed after any operation that modify the access permissions related to the operation *Op* and were performed before *Op*.

That means, if an operation A modifies the result of the access decision for an operation B, the operation A needs to be performed before operation B in every replica. In a weakly consistent system, there is no such guarantee *a priori.*

4. Related work : ACGreGate Model

ACGreGate is an access control model for an Access Control over Antidote. It stands for « Access Control for Geo-replication Gateway ». ACGreGate model was designed as a proof-of-concept that it is possible to come up with sensible access control semantics under causal consistency.

ACGreGate has been designed, among other reasons, to solve the issue of permission replication. Weber et al. [8] show indeed that the relation between on the one hand the causal order and on the other hand the visibility order between two operations is protected in the Causal Consistency[9] model.

4.1 Approach

ACGreGate is a resource-based Access Control model designed between the application layer and the database layer (i.e above the database layer). It intercepts the resource-based requests to the database and forwards or aborts the transaction, depending of the access decision result. It can perform application layer operation based access decisions if specified in the pluggable decision procedure.

To solve the permission replication issue, it stores the Access Control resources in the same database than the objects to provide Transactional Causal Consistency in the access resources and hence ensure the causality between access control operations and database operations. In the previous model, we have : $P \subset Rsrc$.

4.2 API & semantics

ACGreGate uses a new model for access resources management and interpretation, in order to provide Transactional Causal Consistency in permission management. It separates also on the one hand the access decisions making, on the other hand the access decision calls and the execution of the results of the access decisions.

4.2.1 Policies representation

In the ACGreGate model, access resources are represented with a CRDT named **Policy**. This CRDT is only accessible at the Access Control Layer or below. Permissions are stored in the Policy object as a set of statements.

The Policy CRDT is based on Multi-Value Register CRDT, i.e a data container storing the value of every concurrent update. The concurrent values of the Policy object are then merged by computing a new policy to the client. The algorithm to compute the minimal Policy is the following :
- the concurrent set of statements are compared and divided into two subsets : the subsets of DENY statements and the subset of ALLOW statements.
- the intersection set of the concurrent subsets of ALLOW statements is returned.
- the union of all concurrent subsets of DENY statements is returned.

Thus, this conflict resolution algorithm is aimed to make the access decisions use the minimum access matrix (i.e the most restrictive access matrix) in case of concurrent updates. It goes through all the explicit DENY statements possibility to perform the access decision, but only the minimum set of explicit ALLOW statements.

4.2.2 DecisionProcedure

The DecisionProcedure class is a pluggable, application-specific procedure that interprets the Policies statements and performs the access decisions. It does not have any direct link to the database, instead it just provides methods to :
- determine which policies to read in a database
- perform the decisionProcedures, based on the previously requested Policies and the request intercepted.

4.2.3 Access Monitor

The Access Monitor intercepts the database requests and calls the access decisions for these requests and executes the result of the access decision.

- The Access Monitor intercepts the operations sent by the Client to the Database.
- For every operation, it first asks the DecisionProcedure which access resources (i.e Policy objects) to read in the Database.
- Then it reads in the database the needed access resources, parses them and asks the Decision procedure to perform an access decision for the initial request.
- If the access decision result is positive, it transfers the request to the database and transfers its response to the client. If the access decision is negative, it makes the operation return an error.

4.2.4 Policies Mapping

In ACGreGate model, the Policy objects are stored in the same database than the objects, which is one of the conditions to ensure Transactional Causal Consistency for the Access Resources. To provide Transactional Causal Consistency

using the Policy CRDT, another condition is to use one Policy object per subject and per object, instead of one per requested object. Indeed the Access Monitor provides the mapping of the Access Resources by computing the Policy objects keys from the requested subject's key and the requested object's key. The set of statements used in the access decision for an object O is here distributed between different Policy object for every requested subject.

4.3 Issues

4.3.1 Static Access Control System

ACGreGate interprets the Access Control Resources in a hard coded way for a given application and the Access Decision Procedure is the same for every client of an application. For example, the application-layer operation a user can perform have to be defined with the decision procedure.

To provide an Access Resource that is as general as possible, we want to manage Decision Procedure in a **dynamic** way. I.e to store Access Decision Procedure elements as access resources in the database, to change from a hard coded Decision Procedure to a modular Decision Procedure involving pluggable and ditributed decision elements.

4.3.2 Concept of groups

Groups are a subset of subjects, to represent some permissions applying to a set of users. The concept of groups is currently not implementable in this model. To implement it over the ACGreGate model, either the Access Control System has to update the Policy of every subject in a group for each object, either to store every groups policies in a unique file that specifies the policy for all groups. The first approach is not realistic in a geo-replicated database and the second can not ensure the Transactional Causal Consistent model for group management.

5. DAcCoRD model

Among the issues for Access Control Systems, ACGreGate model above Antidote solves the problem of permission transmission, but not the problem of the access control gap, and does not provide features as groups and dynamic access decision procedure, which are needed in order to provide a general, modular Access Control model. To provide these properties, we present the Dynamic Access Control for Replicated Databases (DAcCoRD) model.

5.1 Need

These issues raise the need of a slightly different model, that would enable :
- Operation-based and context-aware Access Control
- Dynamic Access Decision Procedure
- Groups concept

Use case : banking application

For example in a banking Application, the ACGreGate model is not sufficient.
- When a user request to make a financial transaction, the application need to verify some conditions to allow it to perform the transaction, for example some authentication features. However, the operation : « to make a transaction » can not be translated in terms of resource access. In this example the user has the right to *read* its account balance, but not the right to write its balance. Then, it is impossible for it to performa transaction in the resource-based model, that would eventually update its account balance. Therefore, this application needs way to deal with application-level operations.
In the case where the user has suscribed to an offer with limitations, e.g the user's account can not have a negative balance, the Access Control may need to ensure application-layer invariants and therefore to make a preconditional check on the context of the operation.
- If a user wants to upgrade its offer, the application needs to be able to modify the set of permissions of the user, to switch it to a different set. In our example, it may be the limitation of the monthly or weekly total amount of transactions. Therefore, this application needs to be able to define groups, with different permissions, and to change a user from a group to another.

- The banking comapny may also change its offers and modify the previous limitations. To be able to modify the features & limitations of the offers, it needs to have a dynamic way to perform the Access Decision, to store changes in the database.

5.2 Approach

The DAcCoRD model is inspired from both the ACGreGate model and the AWS/Scality S3 file storage interface.
S3 is an interface for file storage currently used in eventually consistent geo-replicated database. Its Access Control provides some features we need to fill the gap between operation- and resources-based Access Control, to enables groups and dynamic Access Decision.

The DAcCoRD model uses two semantics :
- the Resource-based Access Control, designed over ACGreGate and using Access Control Lists (ACL) as Access Resources,
- the operation-based Access Control, using specific access resources and designed to provide the needed S3 features : groups, context-aware Access Control, and dynamic Decision Procedure.

5.2.1 Invariants

Based on the previous work , the needed invariants for an Access Control System are the following :

- ***Default access invariant***

« A a request {*op, subj, obj, contx*} can not be performed before a root authority updates the permissions related to this request to allow the request. »

I.e for every request, before any permissions related is specified, the access decision is negative.

This right has to be given directly or indirectly by the root authority on the object. In the indirect case, this invariant implies that any subject that changed the permissions related to the current request, received the authority to change them from the root authority or a subject to which the root authority has given the appropriate permission.

- ***Revocation invariant***

« As soon as the permission related to a request {*op*, *subj*, *obj*, *contx*} is revoked, that request can not be performed until the related permission is not granted again. »

In a replicated context with transactional causal consistency, the revocation invariant is the following :

« As soon as an update U is visible, in which the permission related to a request {*op, subj, obj, contx*} is revoked, that request can not be performed until another update U' where this permission is granted again is not visible ».

- ***Ownership invariant***

« For every object, there is always at least one subject that has full control over it. »

There are several ways to provide this invariant. Some Access Control Systems are based on the invariant that every file has one and only one owner, which is the root authority for this object. Some Access Control Systems relies on more complex chains of authority transfer[7].

An interesting point here may be that an Access Control System tends to restrict availability of datas. Therefore, strong security guarantees are more likely to be found in a CP context and harder to provide in an AP context.

In the case of Antidote database with the Access Resources of DAcCoRD or ACGreGate, the principle of always computing the minimal access statements set in the case of concurrent updates of an Access Resource tend to reduce the permissions related to an object. We do not have a way in ACGreGate to control this reduction of permissions and therefore to ensure this invariant.

For exemple, two subjects *subjA* and *subjB* have full control on an object *obj*, that is specified in a dedicated CRDT. That object is replicated on at least two replicas *N1* and *N2*. If *subjA* updates the access resource to give the subject *subjB* exclusive control of object *obj*, and concurrently the subject *subjB* transfers the permissions on object *obj* to

subjA in a similar manner. The intersection of both access resource updates is then empty : the resulting access permissions will not keep the ownership invariant.

5.2.2 Domain concept

- ***Approach***

To provide the ownership invariant, the DAcCoRD model features a concept of domain close to the AWS Internet Auhtentication Management (IAM) root account. This domain provides also a root authority and and lifecycles semantics.

The domain is a read-only flag on the buckets that binds them with a root authority that acts like an owner for the resource. This ownership can not be transfered to another root authority (i.e the flag value can not be modified), to prevent the reduction of permissions. Therefore, domain root is a subject that can always perform operations on the object of the domain.

This domain is then unique in the database and can be seen as a partition of the database. A user subject depends of a domain an not perform any operation on resources or users of another domain.

- ***Resource lifecycle***

A « side-effect » of the domain flag in DAcCoRD is that the objects with a flag need then to be initialized. This initialization enables to define lifecycle semantics like « create resource » or « delete resource ». It is useful to prevent users that do not have the permission to create an object.

- ***Invisibility Property***

However, it might be interesting to not have lifecycle semantics. It enables two properties :
- always have an answer when performing an operation on an untouched object,
- performing an operation on an object that doesn’t *exist* in lifecycle terms (i.e that was never touched nor initialized) has exactly the same effect that making a request that returns a negative access decision.

- ***Domain flag***

Therefore, in the DAcCoRD model, to be as general as possible, there is a domain flag on the bucket, but not on the data. Hence, is provides the needed restrictions and invariants, and still enables to use the weak lifecycle semantics on the data. It is still possible to implement lifecycle in the Access Control using the dynamic property of the DAcCoRD model.

5.2.3 Dynamic Decision Procedure

To provide Dynamic and modular access decision procedure, DAcCoRD model requests in the database another type of access resources, like in S3 interface. These resources provide modular statements to add elements in the context to evaluate in the access decision.

5.2.4 Access Control Gap

- Like the ACGreGate model, DAcCoRD model is located between application and database layer.
- Like the ACGreGate model, it intercepts the requests between the client and the database to evaluate the requests with the existing access resources.

- ***Access Resources***

To manage both acess control types, we use the access feature of the AWS/Scality S3 interface[10] : separate access resources for resource-based and operation-based access resource by using two different resources :
- Policies for operation-based access control and dynamic changes in the Decision Procedure. Every user and every bucket are bound to a Policy object to manage operation-based permissions.
- Access Control Lists(ACLs) for resource-based access control. Each object and bucket uses a dedicated ACL object for resource-based permissions.
The redundancy between the two types of resources and the specific access procedure ensures to return the most restrictive access decision.

- ***Decision Procedure***

For each request $\{op, subj, obj, contx\} \in (U \times O \times A \times \Omega)$ with $obj \in bucket$, the set of access resources *pol* is the following set of objects :
- the ACL of the object *obj*
- the ACL of the bucket *bucket*
- the Policy of the bucket *bucket*,
- the Policy of the user $subj \in group$
- if $group \neq \emptyset$ the group Policy

For each request, the Decision Procedure follows the following algorithm :

- is the subject registered in the domain : if no, abort the transaction.

- is the subject the domain root : if yes, return *allow.*

- is there a statement in one access resource of the *pol* set that explicitly denies this request : if yes, return *deny*.

- is there a statement in one access resource of the *pol* set that explicitly allows this request : if yes, return *allow.*

- if none of the conditions above is verified, return *deny*.

The last check provides the default access invariant by denying access when no access resources has been set for this request.

6. Implementation

For implementing this access control, I based my work on the ACGreGate model implementation made by Mathias Weber (TU Kaiserslautern).

6.1 Framework

We only consider a security layer that prevents users to make unauthorized requests. We assume that the application above the security layer authenticates the client and that the client can not access tha Access Control source code. We do not consider :
- interception of Client-Database communications
- any attacks at the database level. We assume indeed that the only way to perform databases operations is the Antidote Java Client.

6.2 Using ACGregate

The DAcCoRD implementation is divided in two parts : the resource-based part and the operation-based part. The resource-based access control uses ACGreGate Policy CRDT and the same key mapping algorithm. However, it uses a different AccessMonitor, that is common with the operation-based access control. This Access Monitor is slightly different from ACGreGate : it uses a different API and a different type of Decision Procedure.

Like in ACGreGate, this Decision Procedure is separated from the database and performs the decision. But it is not pluggable and doesn’t decide which access resources to interpret in the decisions, but notifies the Access Monitor if extra access resources are needed (for example a group Policy). In the contrary, it uses the DAcCoRD Policies to plug ditributed changes in decision Procedure.

6.3 S3 Access features implementation

6.3.1 Domain flag implementation

We implement the domain flag as a :
- Last-Writer-Wins register in every bucket of a domain.
- Key mapping for requesting user Policies. The user Bucket key is computed from the domain name to provent cross-domain operations.

As described in the model, the domain root can create/delete a bucket by setting its domain flag and create/delete users or groups by initializing their user Policy in the user bucket.

6.3.2 Access Resources

The Access Resources are the following :

- Access Control Lists (resource-based Access Resource) implemented as ACGreGate Policy CRDT :
    - object Access Control Lists : bound to every CRDT in a bucket,
    - bucket Access Control Lists : bound to every bucket.

ACLs are stored in different Policy obejcts containing lists of operation a user can perform at the database level from the set : $\{read, write, readACL, write\ ACL\}$. If the operation requested is contained in the ACL, a positive result is sent to the Decision Procedure.

- Policies (operation-based Access Resource) implemented as pluggable AccessDecision stored in CRDT :
    - user Policies : with a set of permission where the subject is the user or groups of users,
    - bucket Policies : with a set of permission where the object is the bucket.

Policies are JSON objects containing a set of Statements objects. A Statement are a tuple $(effect, request)$. The *request* is the object to compare with the request being performed. If they are the same, the *effect* boolean is used in the Decision Procedure.

Both types of resources are managed as object in the Client, and performing an operation creates a CRDT object to handle remote the operation. The request on the access resource use a different API to not recursively call access decisions.

6.3.3 Context management

To provide operation-based access control, DAcCoRD needs information about the current performed operation and the current context when interpreting a Policy. When starting a transaction, the application may send to the Access Monitor arbitrary data, called *context*, of any data type. These data is then checked as specified in the DAcCoRD Policies. This enable to specify context-aware behavior, for example a preconditional check of the IP or of the request date and time.

6.3.4 Dynamically request Policies

In the DAcCoRD model we don't want to separate the interpretation of the Policies and the choice of which Policy to request in the database. The idea is that a Policy might need to request an other one to get additional informations to provide dynamicity. For example, if a user belongs to a group, it is specified in its user Policy. The Access Monitor needs then to request the group Policy, stored as a user Policy and interpret it. This goup may also be a sub-group from a larger one, etc. However, it raises the problem of termination. At this point, I have no guarantee that this process terminates, especially in a weakly consistent replicated context.

For example two admins can update concurrently groups A and B, admin 1 specificating that the group A is a subgroup from group B, and admin 2 specificating that the group B is a subgroup from group A.

For this reason, we use the Decision Procedure architecture of ACGreGate, but enable the Decision Procedure to notify the Access Monitor once to request an extra Policy object. In the group Policy example, it is not possible in DAcCoRD to create subgroups.

6.3.5 Code Architecture

As in ACGreGate, we separate the Access Monitor and the Decision Procedure. We also separate the metadata mapping from the Access Monitor and we provide two clients.

The client is the class that manages the Antidote data, starts and commits the transaction. In the DAcCoRD client, the transaction need a domain name and if needed a *context* data. The client provides a method *loginAsRoot(domain)* to create a new client named *DomainManager* with root authority and permissions. This client can create bucket and classes.

6.3.6 Link between datas & metadata

As in ACGreGate, DAcCoRD stores all the datas and metadatas in different buckets. The name of the Security Bucket and Data Bucket and the keys of metadata are computed by the Key Mapping class.

DAcCoRD uses for the ACL the same mapping idea that ACGreGate : the key mapping function computes the key of the needed ACL from the key of the user performing the operation and the key of the subject.

6.4 Testing implementation

To test DAcCoRD, a few tests are provided with the implementation to test the basic features of the Access Control : explicit deny, default deny, Policy management, ACL management, etc. To test the behavior of the CRDT and the Access Control in a replicated context, tests are being implemented with one DC with several nodes, and several DCs with one node each.

7. Future work and Conclusion

7.1 Layerization of ACGreGate Framework

The first concern after DAcCoRD implementation and testing will be to move it to a different layer. Currently it is integrated to the Java client as a set of librairies, and this client communicates to the database via a Protocol Buffer interface.

We want to separate these librairies from the client by enable them to read in a protocol Buffer interface and translate the unsecure transaction into secure ones. Therefore the Access Control System may act as a Proxy and be separated from the client.

7.2 Access Control overhead

This DAcCoRD model involves many different Access Resources, and therefore need to be compared with existing Access Control implementation, like ACGreGate model.

7.3 User authentication

As specified above, we assumed that the user is authenticated in the application build above the Access Control layer. I would like to implement a user authentication. The authentication in weakly consistent geo-replicated systems may involve a wide range of technologies, to manage sessions cookies in a connection pool for example.

7.4 Encryption

As specified, this Security Layer is basic and not protected against anything at the security layer level. If possible, I want to encrypt the communication between the database and its client. An interesting issue would be to not add too much overhead by using recent encryption algorithm, without reducing the level of cryptographic security.

In parallel to the Database-client communication, we need to encrypt the communication between the different nodes of the replicated database and between the different Data Centers or tag updates with a cryptographic token to ensure that they can not be tampered[11].

Bibliograhy

1. Deepthi Devaki Akkoorath, Annette Bieniusa, Antidote: the highly-available geo-replicated database with strongest guarantees, 2016
2. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain,A. Bieniusa, N. Preguica, and M. Shapiro, Cure: Strong semantics meets high availability and low latency., 0216
3. , Antidote reference platform, 2016, https://github.com/SyncFree/antidote
4. M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski, A comprehensive study of Convergent and Commutative Replicated Data Types, 2011
5. M. Weber, A. Bieniusa, A. Poetzsch-Heffer, EPTL - A temporal logic for weakly consistent systems, 2017
6. M. Y. Becker, C. Fournet, A. D. Gordon, SecPAL : Design and Semantics of a Decentralized Authorization Language,
7. Wobber, T., Rodeheffer, T.L., Terry,, Policy-based access control for weakly consistent replication, 2010
8. M Weber, A Bieniusa, and A Poetzsch-Heffter, Access Control for Weakly Consistent Replicated Information Systems,
9. W. Lloyd, MJ. Freedman, M. Kaminsky and D. Andersen, A short Primer on Causal Consistency, 2013
10. , Amazon AWS S3 Documentation, 2017, http://docs.aws.amazon.com/AmazonS3/latest/dev/s3-access-control.html
11. HTT Truong, CL Ignat, P Molli, Authenticating Operation-based History in Collaborative Systems, 2012