Tiago Miguel Ferreira da Costa

Licenciatura em Engenharia Informática

Access Control in Weakly Consistent Systems

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering

Adviser: João Leitão, Professor Auxiliar,
FCT/NOVA University of Lisbon

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

September, 2016

Access Control in Weakly Consistent Systems

Copyright © Tiago Miguel Ferreira da Costa, Faculdade de Ciências e Tecnologia, Universidade NOVA de Lisboa.
A Faculty of Sciences and Technology e a NOVA University of Lisbon têm o direito, perpétuo e sem limites geográficos, de arquivar e publicar esta dissertação através de exemplares impressos reproduzidos em papel ou de forma digital, ou por qualquer outro meio conhecido ou que venha a ser inventado, e de a divulgar através de repositórios científicos e de admitir a sua cópia e distribuição com objetivos educacionais ou de investigação, não comerciais, desde que seja dado crédito ao autor e editor.

This document was created using the (pdf)LaTeX processor, based in the "unlthesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/unlthesis [2] http://www.di.fct.unl.pt

To my family and friends.

Acknowledgements

I would like to thank my advisor, Prof. Dr. João Leitão and also Prof. Dr. Nuno Preguiça and Phd student Albert Linde, for the opportunity to work with them on this project. Their support made this work possible and helped me enhancing my knowledge. I want to mention the support from all my colleagues in the department. At least, I would like to thank my mother, father, and sister for their support, and friends for their motivation and being by my side in the toughest moments.

This work was partially supported by the European Union through the SyncFree project (grant agreement n°609551) and FCT/MCTES through the NOVA LINCS strategic project (UID/CEC/04516/2013).

Abstract

Eventually consistent models have become popular in the last years in data storage systems for cloud environments, allowing to give users better availability and lower latency. In this model, it is possible for replicas to be temporarily inconsistent, having been proposed various solutions to deal with this inconsistency and ensure the final convergence of data. However, defining and enforcing access control policies under this model is still an open challenge.

The implementation of access control policies for these systems raises it's own challenges, given the information about the permissions is itself kept in a weakly consistent form. In this dissertation, a solution for this problem is proposed, that allows to prevent the non authorized access and modification of data.

The proposed solution allows concurrent modifications on the security policies, ensuring their convergence when they are used to verify and enforce access control the associated data. In this dissertation we present an evaluation of the proposed model, showing the solution respects the correct functioning over possible challenging situations, also discussing its application on scenarios that feature peer-to-peer communication between clients and additional replicas on the clients, with the goal of providing a lower latency and reduce the load on centralized components.

Keywords: Eventual Consistency; Access Control; Replication; Distributed Systems; Peer-to-peer.

Resumo

O modelo de consistência eventual tornou-se popular nos últimos anos em sistemas de gestão de dados nos ambientes *cloud* providenciando aos utilizadores maior disponibilidade e menor latência. Neste modelo, é possível as réplicas fiquem temporariamente inconsistentes, tendo sido propostas várias soluções para lidar com esta divergência de estado e garantir a convergência final dos dados. Contudo soluções para definir políticas de controlo de acessos neste modelo são ainda muito limitadas.

A implementação de políticas de controlo de acessos para estes sistemas levanta desafios próprios, dado que a informação sobre permissões tem de ser ela própria mantida de forma fracamente consistente. Nesta dissertação propõe-se uma solução para este problema prevenindo o acesso e modificação não autorizada dos dados.

A solução proposta permite modificações concorrentes das políticas de controlo de acesso, garantindo a convergência das mesmas enquanto são usadas para efetuar controlo de acessos aos dados associados. Nesta dissertação apresentamos uma avaliação inicial da solução desenvolvida demonstrando que esta permite efetuar o controlo adequado sobre possíveis situações problemáticas, existindo também um estudo da sua aplicação em ambiente que incluem comunicações par-a-par entre clientes e replicas adicionais nos clientes, que visam oferecer uma menor latência e reduzir a carga nas componentes centralizadas.

Palavras-chave: Consistência eventual; Controlo de acessos; Replicação; Sistemas Distribuidos; Sistemas par-a-par.

Contents

List of Figures **xv**

List of Tables **xvii**

1 Introduction **1**

1.1 Motivation 3
1.2 Problem Description 4
1.3 Document Organization 6

2 Related Work **9**

2.1 Access Control 9
2.1.1 Access Control List (ACL) 10
2.1.2 Role-Based Access Control (RBAC) 11
2.1.3 Attribute-Based Access Control (ABAC) 12
2.1.4 Capability-Based Access Control 13
2.1.5 Divergence Issues for Weakly Consistent Replication 13
2.1.6 Access Control Model for Distributed Collaborative Editors . . . 15
2.2 Data Storage Systems 16
2.2.1 Consistency 16
2.2.2 Conflict Resolution techniques 17
2.2.3 Examples of Data Storage Systems 20
2.3 Peer-to-Peer 21
2.3.1 Overlay Networks 22
2.3.2 Example peer-to-peer overlay networks 25
2.3.3 Access Control Basic requirements on a P2P system 25
2.4 Geo-Replicated systems that use some sort of access control 26
2.4.1 Data storage systems with eventual consistency 26
2.4.2 Systems without mutual trust 27

2.4.3 Systems with causal consistency . . . 27
2.5 Summary . . . 28

3 Theoretical Model 29
3.1 Convergence on a Weakly Consistent Model . . . 30
3.1.1 System Model . . . 32
3.1.2 Policies and Trust Model . . . 34
3.2 Access Control Semantics . . . 37
3.3 Inconsistency Challenges . . . 41
3.3.1 Concurrent update on the access control policies . . . 42
3.4 Client to Client Communication Model . . . 43
3.5 Summary . . . 46

4 Implementation 47
4.1 Data Object . . . 48
4.1.1 Applying Access Control to the Original Data Object . . . 50
4.2 Concurrent Updates Detection . . . 53
4.3 Restrictive Minimum Permissions . . . 54
4.4 Operation Ordering Challenge . . . 54
4.4.1 Without Causality . . . 55
4.4.2 With the Use of Causality . . . 55
4.4.3 Causality *vs* No Causality . . . 56
4.5 Summary . . . 57

5 Evaluation 59
5.1 Correctness Tests . . . 60
5.1.1 Sequential Operations . . . 62
5.1.2 Disordered Operations . . . 63
5.1.3 Concurrent Operations . . . 65
5.2 Overhead . . . 66
5.3 Summary . . . 69

6 Conclusion 71

Bibliography 73

A Appendix 77

List of Figures

1.1 Geo-Replicated system with access control 5

2.1 Access Control List example [7] 10
2.2 Role-Based Access Control example [9] 11
2.3 ABAC Scenario [11] 12
2.4 Concurrent policy (P') and data (U') updates leading to permanent inconsistency 14
2.5 Causal consistent[19] 17
2.6 CvRDT: integer+max [29] 19
2.7 Structured P2P [39] 24
2.8 Representation of a Distributed Hash Table [40] 24
2.9 Unstructured peer-to-peer [39] 25

3.1 Geo-Replicated System Example 30
3.2 Inconsistency of data 30
3.3 System trust 35
3.4 Possible data inconsistency 35
3.5 Violation of access control policies case 1 41
3.6 Violation of access control policies case 2 42
3.7 Direct Client Communication Model 44
3.8 Client Communication with Centralized Component Request 44

5.1 Message Size on Data Updates 67
5.2 Message Size on Policy Updates 68

List of Tables

5.1 Operations: Sequential Operations Test 62
5.2 Result: Sequential Operations Result 62
5.3 Operations: Sequential Operations Test with Revocation 62
5.4 Results: Sequential Operations Test with Revocation 63
5.5 Operations: Disordered Operations 63
5.6 Results: Disordered Operations 63
5.7 Operations: Concurrent Modification of Policies Test 65
5.8 Results: Concurrent Modification of Policies Test 65

Chapter 1

Introduction

In recent years there has been an increase in the popularity of web applications and web services which became an extremely relevant industry. Taking a closer look, a large number of those applications are centered on users. By user centered, we mean applications in which users are both the major producers and consumers of content, and where the major role of the application is to mediate interactions between them. Facebook and Twitter are among some of the most popular examples, while other examples can be found in collaborative editing tools, such as Google Docs or the online Office 365, games, and chat systems such as the Facebook chat service.

It has been demonstrated that it is essential to provide low latency for web applications, as failing to do so has a direct impact on the revenue of applications providers [17]. When there is a large number of users using a web service, scattered across the World, there will be latency issues for those that are further away from the centralized component. One simple example is a user from Europe trying to access a service in the United States.

Due to this, some of the most important factors for the success of such applications are high availability and low latency of operations provided to the client. A common strategy to reduce the latency is to resort to geo-replication, meaning that the service provider has servers and its data distributed (by replicating or partitioning data) across multiple data-centers, scattered throughout the world, so that content can be served to all users with the lowest possible latency. This implies that data is distributed among

servers that are connected by high latency connections that might not be reliable, meaning that, for instance, a service provider with clients in United States and Europe, would have servers in the United States and in Europe, as to lower latency for all users, independently of their locations. This ensures low latency and fault tolerance (as users can always fail over to the other data-center if their local data-center becomes unavailable). There is however a challenge that arises in this environment, formally captured by the CAP Theorem [31] , which states that it is impossible for a distributed computer system to simultaneously provide strong consistency (all nodes see the same data at the same time) and availability (every request receives a response about whether it succeeded or not) in an environment where network partitions can happen. Due to how the Internet infrastructure operates, partitions due to node failures have to be considered as they will eventually happen, so there is the question of which to choose between strong consistency or availability. The common practice nowadays is to privilege availability, so most practical systems offer only some form of weak consistency, and in this work we tackle additional challenges that arise in this context.

A common used model of weak consistency is eventual consistency. This model guarantees that, if no updates are made to a given item, eventually all accesses to that item will return the last written value [36] . In this model there may be times of inconsistency among the replicas of a data object, as replicas might be temporarily in different states. An eventually consistent system can return any value before it converges. This allows these systems to always, even during network partitions, perform operations over data. Eventual consistency however, might not be enough. Consider the following example: user A writes to a page and user B answers. Due to network latency user C might observe B's answer before A's initial post. This shows that while the consistency model is not violated, it can lead to unexpected behavior from the standpoint of the users. On the other hand, this model provides high availability (every request receives a response about whether it succeeded or not).

Most applications usually keep sensitive information about users, which they do not want to be obtainable by other users without their authorization. With this, the access control over such information is a very important aspect in every application with multiple users, which requires the existence of an access control mechanism that enables the definition of access control policies that specify who has access to read or modify each part of the existing information at each moment. These policies can also change over time, requiring access control mechanisms able ensure that the policies become active as soon as the moment when they were modified, as to disallow unauthorized operations.

Access control is a well studied subject in systems that guarantee strong consistency, where multiple models and mechanisms to restrict the access to information [8, 11, 12, 14, 27, 28] have been proposed. However the same is not true in systems that only guarantee weak consistency. Under strong consistency models there is a total order between operations that are executed. For example, to control the access to information in a social network, ordering the operations that modify a list of friends and the access to the information of a user, make it possible to easily guarantee that a user will be unable to read information about another user after he has been removed from his list of friends.

Unfortunately, in systems that only provide weak consistency, there is no total order between operations executed in the system. With this, different replicas may execute operations in different orders. This fact raises new challenges for the models of access control as for the mechanism that implement (and enforce) such models.

In this dissertation we present a model of access control for data storage systems that use weak consistency and proposes mechanisms to implement this model. Similar to the solution proposed by Webber et. al. [38], our solution allows concurrent modification of data and policies that control the access to that same data, defining in a deterministic way the behavior of the system. However, opposite to this previous approach, we also propose a new mechanism of implementation of the defined model that allows its use in data storage systems that don't provide any form of causal consistency, which is common among the largest fraction of NoSQL systems - e.g. Cassandra, Riak. Also, we discuss and implement access control mechanisms under this model for emerging distributed architectures that enable user applications to communicate directly between them and have themselves local replicas of the application state [20], which allows lowering the dependency on centralized components for web applications. Our solution allows enriching such applications with access control, without losing too much introduced by these architectures, meaning without depending too much on the centralized component to enforce access control.

1.1 Motivation

In classical architectures, the server component is fully responsible for many correctness and security aspects of applications. In particular, a storage system is fully responsible for maintaining all data and ensuring the durability of such data. Moreover, the server component mediates operations performed over this data, controlling potential state

divergences that might arise due to concurrent operations issued by users. Access control and integrity are also fully delegated to this centralized component in centralized architectures.

For Geo-Replication the same applies but with multiple centralized components. The data would be replicated between nodes and those nodes still need to be responsible for many correctness and security aspects of the applications. These correctness aspects have to be maintained individually by each site, but also between them. Due to the use of eventual consistency, the state that exist in each replica, including meta-data structures used to perform access control tasks among others, will diverge. This is something that has to be addressed in order to provide access control and integrity of data.

Access control is what defines the permissions that something or someone has on a given object or resource. There are multiple models and implementations of access control, such as the access control list (ACL) in UNIX.

In a typical client-server architecture that feature access control, where each request the access control component is verified to check the permissions and evaluate if the request is valid. When a revocation of the writing permission of a user and subsequent attempt to update data occur, they must be handled in a sequential way, meaning that the update will be checked with the new access control modifications and be refused.

In a Geo-Replicated system with eventual consistency, adding the access control layer, brings issues evaluating current access control policies. This happens because of the possible temporary inconsistency of meta-data structures used to perform access control, since in one server the permissions may have been revoked and in the other the old permissions are still active allowing updates and other operations to be performed when they should have been disallowed.

The peer-to-peer paradigm mentioned earlier will inherit the issues of the geo-replication since it also uses the multiple centralized components, but also allows for direct communication between clients. This communication will also need to have an access control mechanism in place across all the clients, adding a new set of challenges to be addressed.

1.2 Problem Description

As mentioned earlier, the geo-replication of data already leads to a break up with strong consistency. In the classical client-server model, the updates were made on the same place meaning that multiple users would be updating data in a ordered way, where

the first update to reach would be the one that would win, and data would be always consistent.

In a geo-replicated system with eventual consistency this is no longer a reality, and with this adding a layer of access control becomes more challenging.

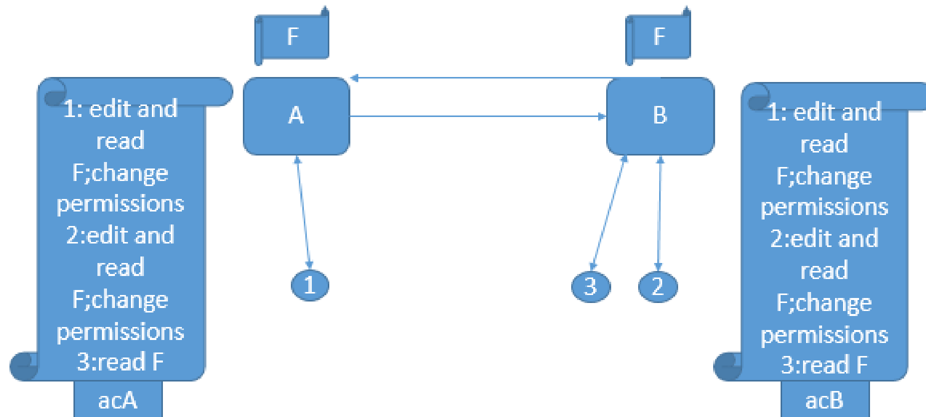


Figure 1.1: Geo-Replicated system with access control

Imagine the example illustrated in Figure 1.1. We have two servers with replicated data and access control (A and B), and three clients. The two servers start with the same file F, being in this way consistent in data and also in the access control (acA, acB). With eventual consistency some issues may appear given the presence of access control mechanisms.

If client 1 removes the read permission to client 3, there will be the need for that request to arrive to B, but because of the use of eventual consistency there will be no guarantee when that revocation will arrive, meaning that client 3 will still be able to see file F given the fact that the access control in B is outdated. This wouldn't happen on a single-server system since the server would take care of the order of actions performed by different clients ensuring a behavior consistent with that ordering.

In the previous case it wouldn't be problematic since 3 would only be able to see a file equal to the one that existed before when he had permissions, since no edit had been made, but it is possible to imagine a situation where user 2 makes changes to the document that user 3 shouldn't be able to see, however because of the eventual consistency on the access control he will still be able to observe the effects of such operations, which might be an undesirable outcome for this execution.

This leaves us with a set of issues that occur when there are concurrent writes, reads

and changes on the permissions, in a geo-replicated environment with weak consistency. We can also assume malicious behaviors, for example, client 3 already knows that his permissions have been revoked but he submits a request with a time-stamp with a value such that his operation is ordered by the system before the actual change in permissions (assuming the use of time-stamp by the centralized components).

When one adds direct communication between clients, the issues from geo-replication will be inherited and some additional challenges arise, since in such a scenario, data can also be present in the client side and updates over the data can be issued directly among clients. This leads to the need of having access control mechanisms to be present in the client side. All of this needs to be maintained using eventual consistency, where updates from a user have to be propagated for other users and centralized components while ensuring that access control policies are enforced in a meaningful way.

The focus of this thesis is to study how to address these challenges in the context of both geo-replicated systems and in systems that leverage an extension to the new browser-to-browser paradigm introduced in [20].

Publications:

Part of the results in this dissertation were published in the following publications: **Controlo de Acessos em Sistemas com Consistência Fraca** Tiago Costa, Albert Linde, Nuno Preguiça e João Leitão. Actas do oitavo Simpósio de Informática, Lisboa, Portugal, September, 2016.

1.3 Document Organization

The remainder of this dissertation is organized as follows:

Chapter 2 describes the related work. Existing access control models, covering existing access control methods where we further discuss how a weakly consistent replicated system can affect the enforcement of access control policies. This chapter also discusses web based data storage, where different consistency models and ways to solve some consistency issues are discussed. Relevant existing examples are also presented. Finally, this chapter discusses existing peer-to-peer technologies and solutions, which are relevant to understand how to extend our work to hybrid architectures supporting web applications.

Chapter 3 describes the model of the system in which the access control was integrated, the semantics the access control should follow and what should a user see in

the system, the trust model and the existing challenges for adding access control on a system operating under weak consistency.

Chapter 4 describes the details of the implementation and how the solutions discussed on Chapter 3 were integrated with practical examples and a more concrete description.

Chapter 5 presents the evaluation of our system and the results compared to an approach without our security guarantees.

Chapter 6 concludes this dissertation also discussing future work.

Chapter 2

Related Work

This dissertation addresses the challenges of enforcing access control to distributed data systems that use the eventual consistency model, and extending such solutions to distributed data storage systems that leverage on peer-to-peer communication allowing clients to communicate directly (and have their own replicas of data). The following sections cover the main aspects of these fields, in particular:

Section 2.1 existing access control models are discussed and compared. It also reviews some of the existing proposals in the context of practical systems.

Section 2.2 web based service providers are discussed, in particular focusing on the challenge of storing and accessing data, and maintaining that data given the trade-off between consistency and availability.

Section 2.3 existing peer-to-peer technologies are presented and compared.

Section 2.4 existing geo-replicated systems that use some form of access control are compared to the objective of this dissertation.

2.1 Access Control

Access control is a mechanism for a system to know and enforce the execution of a given action by some entity (usually a user or a program acting on the behalf of a user) is

allowed. In the context of access control usually one refers to a **principal** as someone (or something) that interacts with a system through actions, and a **resource**, as an object that the principal manipulates through the execution of actions.

Access control mechanisms and policies keep track of which principals can access existing resources in the system, for example a user to access a file on a computer it is the access control that will verify the user is able to access that file and what type of actions can I perform over it (e.g. read, write, execute).

In the domain of access control, one can also think in terms of policies. Policies are statements that will be evaluated to check if the user has access, those policies can vary in complexity, they can either be small and evaluate only one attributed like "user has role X", or more complex with more attributes, where for example "user has role X, is in local Y, in date Z".

Access control only restricts the operations of legitimate users, meaning that it is only a partial part of a secure system, since in most systems it needs an authentication service to prove the legitimacy of the user.

2.1.1 Access Control List (ACL)

	Page/Namespace	User/Group	Permissions[1]
#1	*	@ALL	○ None ○ Read ○ Edit ◉ Create ○ Upload ○ Delete
#2	*	bigboss	○ None ○ Read ○ Edit ○ Create ○ Upload ◉ Delete
#3	devel: *	@ALL	◉ None ○ Read ○ Edit ○ Create ○ Upload ○ Delete
#4	devel: *	@devel	○ None ○ Read ○ Edit ○ Create ◉ Upload ○ Delete
#5	devel: *	bigboss	○ None ○ Read ○ Edit ○ Create ○ Upload ◉ Delete
#6	devel: *	@marketing	○ None ◉ Read ○ Edit ○ Create ○ Upload ○ Delete
#7	devel: funstuff	bigboss	◉ None ○ Read ○ Edit ○ Create ○ Upload ○ Delete
#8	devel: marketing	@marketing	○ None ○ Read ◉ Edit ○ Create ○ Upload ○ Delete
#9	marketing: *	@marketing	○ None ○ Read ○ Edit ○ Create ◉ Upload ○ Delete
#10	start	@ALL	○ None ◉ Read ○ Edit ○ Create ○ Upload ○ Delete

Figure 2.1: Access Control List example [7]

An access control list (ACL) Figure 2.1 is the most simple materialization of access control, it fundamentally relies on a list of permissions attached to each individual resource. This list is a data structure containing entries that specify individual users or groups (Principals) and explicit permissions to that specific object (Resource). It is basically a list with entries that defines the permissions that apply to an object and its properties. This allows a fine-grained access control over the permissions of principals.

When access control lists were first introduced, they were more effective as systems had a low number of principals where each principal had different rights. Modern systems evolved to have a large number of users, leading to a high amount of entries. If there was the need to make changes in various principals one would have to change each one of them individually. Nowadays it is possible to join users in a group and treat that group as a principal, assigning or revoking rights to all elements (i.e. users) of the group at once.

In an ACL implementation it is easy to find the set of all principals who may read a file, but it might be difficult to find the set of all files that a subject may read. This is because the access control list is stored for each resource containing the principals and their rights to that file individually.

2.1.2 Role-Based Access Control (RBAC)

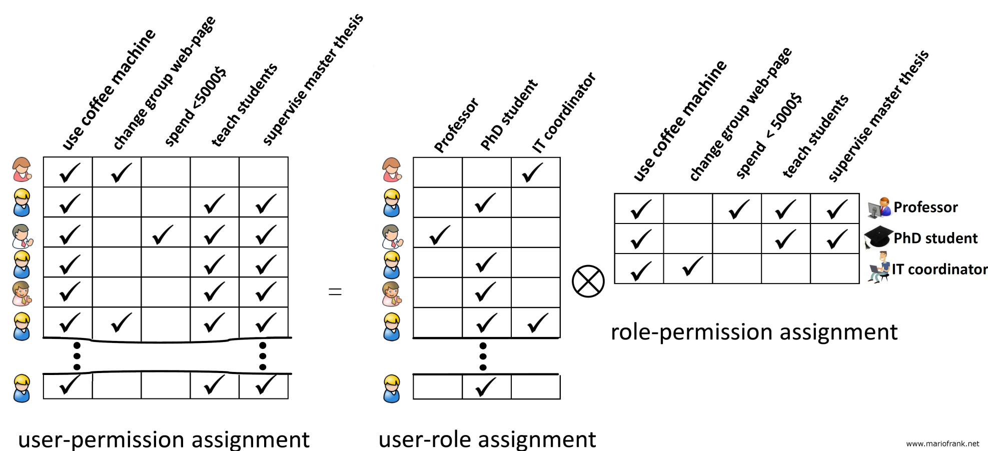


Figure 2.2: Role-Based Access Control example [9]

In large scale systems, security policies are dynamic. Access rights need to change as the responsibilities of principals change. This can make management of rights difficult, as when a new user joins the system, the appropriate rights for that user must be established, and when a user changes job functions, some rights should be removed, while others have to be added. In a broad example, we can think of an hospital, where there are multiple doctors, nurses, chief executives, patients, etc... Each doctor should have the same access rights as the others, as well as nurses among them. If a nurse changes functions to doctor, there is the need to change all of her permissions so that they are the same as the remaining doctors. This increases the complexity on changing rights

given a multiple number of principles where most of them are in theory in real world function groups where all should have the same permissions.

Role-based access control, as illustrated in Figure 2.2 addresses this problem by changing the underlying principal resource model. In Role-Based Access Control, principals are classified into named roles (user-role assignment). A role has a set of actions and responsibilities associated with a particular working activity. Instead of an access control policy being a relation on principals, resources, and rights, a policy is a relation on roles, resources, and rights. For example, the role "professor" might be assigned the right to "supervise master thesis". Principals are now assigned to roles, where each subject may be assigned to many roles and each role may be assigned to many principals. Roles may also be hierarchical, for example, the role "professor" may have all the rights that a "phd student" does, and so forth.

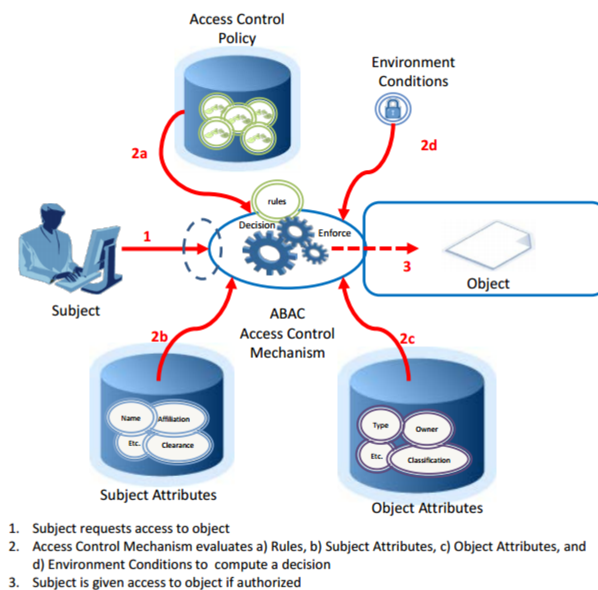


Figure 2.3: ABAC Scenario [11]

2.1.3 Attribute-Based Access Control (ABAC)

Attribute-Based Access Control, (Figure 2.3), is an access control technique where the principals requests to perform an operation on given resources are granted or denied based on assigned attributes to the principal, assigned attributes of the object, environment conditions, and a set of policies that are specified in terms of those attributes and conditions. This might be a more logical way to perform access control, since there is

the evaluation of rules against attributes to check the permissions. In Figure 3.3 it is possible to see a simple ABAC scenario, where the decision to give or deny access to an resource (object) is given by having in consideration the subject (principal) attributes (2b), the object (resource) attributes (2c), the access control policy (2a) and environment conditions (2d).

It is possible to look at ABAC as a broader access control technique when compared to ACLs and RBAC, since ACLs work on the attribute of "identity" and RBAC on the attribute "role", while ABAC is the concept of policies that express complex rules that can be evaluated over many different attributes.

2.1.4 Capability-Based Access Control

Capability-Based Access Control [34] is an access control technique in which principals have tokens which can be shared and that will give permission to access an entity or object in a computer system. An intuitive example for this mechanism of access control is a key that opens the door of an house, or an access card in an hotel that allows you to enter your room, in both these cases it is not only possible to more people to share your permissions, since they can also have keys or cards that also allow the access but also to a user to transfer its capability to another user (by handing off the key).

A capability is implemented as a data structure that contains an Identifier and an Access Right (read, write, access, etc).

A possible analogy with the real world may be the way a Bank works. With a ACL the Bank would have a list of the clients of the bank and the persons that could access each bank account. If a user wants to give permissions to another person there was the need for she to go to the bank and ask to add the permissions for that new person. Meanwhile a capability based approach, there would be a key to each safe, representing the bank account, and the user would only need the key to have access to the safe, meaning that if she would like to give access to another person she would only need to make a copy of the key and give it to the other person which would then use that same key to enter in the safe.

2.1.5 Divergence Issues for Weakly Consistent Replication

When we are using a system with replicas that relies on weakly consistent replication some problems may arise. If the authorization policy can be temporarily inconsistent, any given operation may be permitted at a particular node and yet denied at another, and without a careful design, permanently divergent state can be a result of such a

system. This issue is more serious when access policies are being modified, as the probability for divergence is higher during such periods.

We can have a system where all the nodes trust each other (for instance they are all controlled by the same entity), in this case the access control policy for allowing an update U can be enforced independently by each node, even though there might be transient variants of the policy in each of the nodes. Because the nodes trust each other to enforce policies, they will never permanently disagree about which operations have been accepted, since once an update is admitted, no further checks are required, although it is possible that the most recent policy is not used for the access control decision of some operations. However one can assume that such nodes will not disregard the current policy to allow a principal to execute some operation that was not allowed.

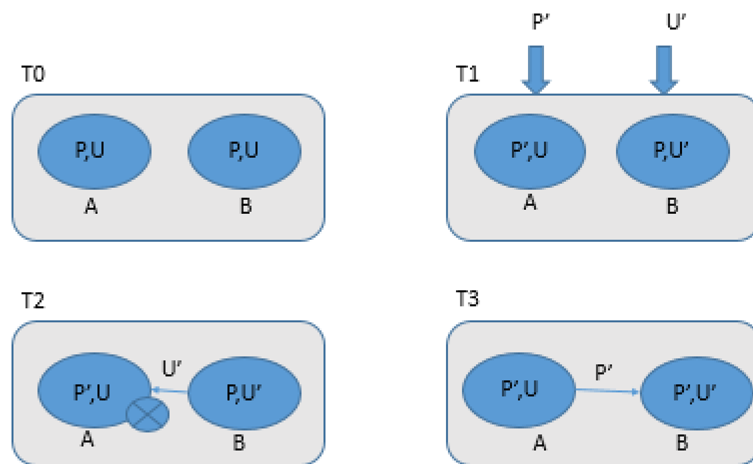


Figure 2.4: Concurrent policy (P') and data (U') updates leading to permanent inconsistency

In a distrustful system (i.e, a system where the individual components might belong to different administrative domains, and hence might deviate in the execution of a distributed protocol), there is the need to explicitly address consistency issues, as illustrated in Figure 2.4. In this example we start at T0 with two replicas (A and B) with the same Access control policy (P) and the same data (U). In T1, two concurrent updates, one of policy (P') in replica A and another of data (U') in B are issued. Assuming that the control policy P allows the update U' and that policy P' doesn't allow it. In T1 it will be possible to change the data to U' in B since the local control policy P allows that operation over that data item, while in A the policy changes from P to P'. In T2 there will be an attempt to propagate data U' to the node A, but since P' doesn't allow the

update U,' node A won't accept the update. In T3 A will propagate its control policy to B, meaning that A will stay with (P',U) while B with (P',U'). This leads to a scenario with a permanent state of inconsistency.

2.1.6 Access Control Model for Distributed Collaborative Editors

Distributive Collaborative Editors (DCE) allow users to simultaneously modify shared documents, one of those examples is Google Docs. To ensure availability shared documents are replicated on the site of each participant user of the group, and updates to that file are made locally and afterwards propagated to the other members of the group. This editors need to provide *high local responsiveness*: the system should be as responsive as single-user editors, *high concurrency*: it should be possible for users to modify any part of the shared document concurrently and freely, *consistency*: all users should eventually see the same state, meaning that there is a need of a convergence between all copies, *decentralized coordination*: there should be no single point of failure and *scalability*: so that any number of users can join or leave a group.

Controlling access in such systems is a challenging problem, since they need to allow dynamic access changes and need to provide low latency access to shared documents without violating the properties discussed above. Adding an access control layer, the high responsiveness is lost because every update must be granted by some authorization coming from a distant user (as a central server). The problem consists in the latency added by access control-based collaborative editors due to the use of one shared data-structured containing access rights that is stored on a central server. This way the controlling of access will consist in locking the data-structure and verify whether the access is valid.

Work presented in [13] has been made to try and tackle this issue, consisting in adding another copy to the user side consisting on the access data-structure. This way all the users from a group get locally the shared document and the access data-structure. When users want to manipulate a shared document, this manipulation will be granted or denied by verifying only the local copy of the access data structure, meaning it will not need to make an access to a centralized component. The main drawback in this work is restriction on who can make modifications to the access control data-structure, since only one user, called administrator, will be able to modify the shared access data-structure. Those updates locally generated by the administrator are then broadcast to other users. This solution removes the complexity relative to the occurrence and resolution of conflict due to different ordering of concurrent modifications to the access

control data, but leaves an important drawback of only one user performing access control modifications. There are still possible security holes because of the absence of safe coordination between document's and access the access control data-structure's different updates, using an optimistic approach that tolerates momentary violation of access rights but ensures a final valid state to the stabilized access control policy.

2.2 Data Storage Systems

As discussed previously, data storage for web services commonly resorts to geo-replication, as web based services benefit from storing client data on geographically distributed data centers, to provide a lower access latency, improved bandwidth, and availability.

There is however a challenge that arises in this environment, which has been captured by the CAP theorem. The CAP theorem, as mentioned in the Introduction of this document, states that it is impossible for a distributed computer system to simultaneously provide strong consistency and availability in an environment where network partitions can happen. This implies that a geo-distributed system must either sacrifice availability or strong consistency.

2.2.1 Consistency

Strong consistency: In a strongly consistent system, if there is a write operation that terminates, the next successful read on that key is guaranteed to show the effect of the previous write operation, meaning that a client will never see out-of-date values. The problem with the strong consistency is the trade-off it makes with availability, since a distributed system providing strong consistency may come to a halt if nodes become unavailable due to a fault or a network partition as all write operations must be performed in a coordinated fashion across all replicas. This may let the service unavailable for the user during a long time, resulting in a bad user experience. Strong consistency however minimizes the inherent challenges related with dealing with consistency of data, since data will be (always) consistent across all the sites.

Eventual consistency: An eventual consistency system is used to achieve a high availability. The system guarantees that if no new updates are made to a given data item, eventually all accesses to that item will return the value of the last update [36]. In this case there may be times of inconsistency among the replicas of a data object, since it does not make safety guarantees, an eventually consistent system can return any value before it converges. This enables these systems to, even during network partitions, always

serve read and write operations, following the CAP theorem sacrificing the consistency and promoting availability.

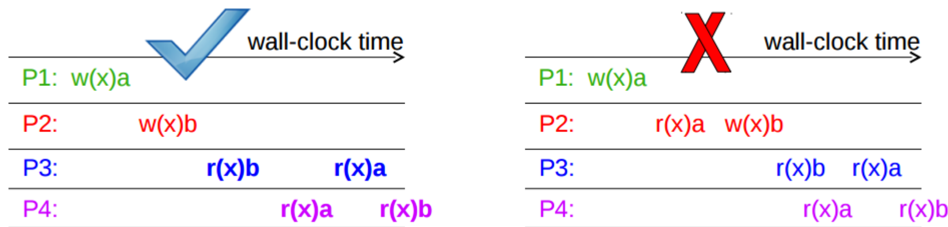


Figure 2.5: Causal consistent[19]

Causal consistency: A system provides causal consistency if any execution is the same as if all causally-related read/write operations were executed in an order that reflects their causality. Concurrent operations that are not causally related may be seen in different orders by different clients. When a client performs a read followed by a write, the first operation is said to be causally ordered before the second, because the value stored by the write may depend on the result of the (previous) read operation. Two write operations performed by the same client are also considered causally related in the order they were performed. In Figure 2.5, we have a representation of a causal consistency system on the left, and on the right a system that is not causally consistent since, w(x)b (this represents a write operation on the date object identified by key x with the value of b) is causally-related on r(x)a (this represents a read operation on the data object identified by key x that returns the value a) on P2, which is causally-related on w(x)a, therefore it must be enforced w(x)a w(x)b ordering, but P3 violates that ordering.

Using eventual and causal consistency comes with the cost of state divergence, since only strong consistency guarantees data consistency (i.e, no divergence) at all times. Since there may be some cases of state divergence, some conflict resolution techniques must be used, such as the ones discussed in section 2.2.2.

2.2.2 Conflict Resolution techniques

As mentioned earlier, applying weaker models of consistency may leave the the system in a state of divergence. In order to ensure replica convergence, a system needs to exchange versions or updates of data between servers (anti-entropy) and choose the appropriate final state when concurrent updates have occurred (reconciliation). For a

system to be able to return to a point of consistency across all replicas, some conflict resolution techniques can be employed, including:

Last Writer Wins: In the last writer wins technique the idea is that the last write based on a node's system clock will override an older one. This is trivial using a single server, since it only needs to check when the writes came and apply them in order, but using multiple nodes where clocks may be out of sync may be an issue. Choosing a write between concurrent writes in this case can lead to lost updates.

Programmatic Merge: Programmatic merge consists in letting the programmer decide what to do when conflicts arise. This conflict resolution technique requires replicas with to be instrumented with a merge procedure, or to expose diverging states to the client application which then reconciles and writes a new value. With this technique the final write will always be the one decided by the programmer, meaning that the most important data for the programmer will be kept.

Commutative Operations: Commutative operations are as the name hints, operations where changes in the order will not change the final result. If all operations are commutative, conflicts can be easily solved since, independently of the order in which operations are received (and applied) in each node, the final result (i.e, state) will always become the same. Commonly used techniques based on the commutation of operations are:

OT, Operational Transformation. OT was originally invented for consistency maintenance and concurrency control in collaborative editing of plain text documents. The idea of OT is to transform parameters of executed operations so that the outcome is always consistent. Given a text document with a string "abc" replicated at two sites with one user on each site, and two concurrent operations where user 1 makes a request of inserting character "x" at position "0" (O1) and user to makes makes a request for deleting the character "c" at position "2" (O2). This request will reach first the site they are using, and because of latency only after some time be propagated to the other server. Because of this, in site 1 we will have first operation O1 executed, and only after O2, while the opposite will happen on in site 2. With this the result in site 1 will be "xab", while on site 2 the result will be "xac", staying in a state of divergence. Using OT we will be transforming the operations to solve this problem, the delete is transformed to

increment one position and the insert can remain the same. Both outcomes become "xab", independently of the order in which operations are applied.

Operational Transformation has been extensively studied, especially by the concurrent editing community, and many OT algorithms have been proposed. However, it was demonstrated that most OT algorithms proposed for a decentralized OT architecture are incorrect [24]. It is believed that designing data types for commutativity is both cleaner and simpler [29].

CRDT, Convergent or Commutative Replicated Data Types. CRDTs are replicated data types that follow the eventual consistency model. An example of a CRDT is a replicated counter, which converges because the increment and decrement operations commute naturally. In these data types, there is no need to synchronisation, an update can execute immediatly and return the reply to the client, unaffected by network latency between the replicas of the data object. The replicas of CRDT are guaranteed to converge to a common state that is equivalent to some correct sequential execution by design [29]. CRDTs can typically be divided in two classes:

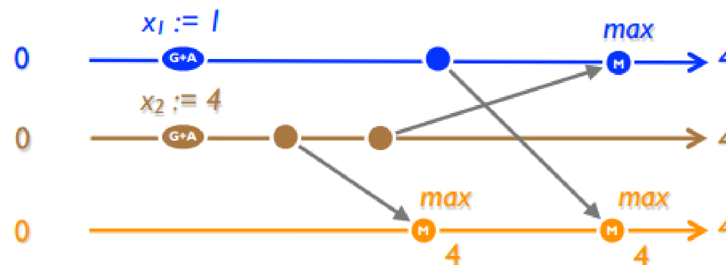


Figure 2.6: CvRDT: integer+max [29]

State-Based CRDT, state-based Convergent Replicated Data Type (CvRDT. In state-based replication, an update occurs entirely at the source, and only after the local execution, there is synchronization between replicas. This synchronization is achieved by transmitting the full state of the CRDT (value and any internally maintained meta-data) between replicas. It is possible to see in the example in Figure 2.6, where we have a CvRDT integer with a max function, the update if first made at it's origin and then the whole resulting object (the integer) is sent to the replicas where it will converge, giving us the same final state across all replicas.

Operation-Based CRDT, operation-based Commutative Replicated Data Type (CmRDT).

CmRDT are based on operation commutation, in this case the operation is executed at the source and after this execution, that operation is propagated to all the remaining replicas. This is possible because in the operation-based class, concurrent operations commute. Operation-based replication requires reliable broadcast communication delivery with a well-defined delivery order, such as causal order between operations. This is important to avoid a replica to evolve to states that shouldn't exist.

In summary we can differentiate CvRDT and CmRDT by the information that is propagated between their replicas. In the case of CvRDT the update is applied and the object that results from executing that function is then sent and merged, while on CmRDT it is the function that is propagated. A simple example could be asking someone to raise their hand, in the CvRDT type we would apply the function of raising someone hand and then send the entire object person (with the hand raised) to merge in the other replicas, while in the CmRDT type we would just send the function that contains the instruction to raise the hand.

2.2.3 Examples of Data Storage Systems

Spanner: [4] Spanner is a globally-distributed system that uses the Paxos algorithm in order to replicate data across a large number of data centers providing strong consistency. One of the replicas is elected as the Paxos leader for the replica group, that leader is the entry point for all transnational activity for the data objects managed by that group. Also the system automatically re-shards data across the machines when the number server or amount of data changes, making it a scalable system that can perform under high load balance and with machines failing and going back up.

Having strong consistency means all Spanner transactions are globally ordered, being assigned hardware assisted commit time-stamp.

With this we can verify that Spanner is a scalable (it balances the load being able to deal with an increase of requirements), it multi-version (so it can provide strong consistency) and is globally-distributed.

Dynamo: [6]

Dynamo consists in the opposite of Spanner, since it aims to provide an high level of availability sacrificing consistency under certain scenarios to achieve this. With the use of weak consistency the problems of inconsistency of data between

replicas is introduced, being it solved by exposing the data consistency issues and reconciliation logic to the developers. In Dynamo data is partitioned and replicated using consistent hashing.

Dynamo also employs a gossip based distributed failure detection and membership protocol.

Cassandra: [18]

Cassandra was also made in order to be able to provide high availability and no single point of failure. It uses consistent hashing (like dynamo) to partition data across the clusters. To achieve high availability and fault tolerance it replicates the data across multiple data centers making it possible to continue to provide it's read and write operations even existing some points of failure. In relation to the replication methods the main difference between Cassandra and Dynamo is the fact that Cassandra receives a selectable replication factor that will define on how many nodes each data item will be replicated.

Riak: [16] is a distributed NoSQL key-value data store that supports high availability by giving the flexibility for applications to run under strong or eventual consistency, using quorum read and write requests and multi-version concurrency control with vector clocks. Eventual consistency in Riak uses CRDTs at its core. Partitioning and replication is done via consistent hashing.

Antidote: [1] Antidote is a strongly, eventually consistent storage back end for extreme scale cloud services and applications. It support Intra and Inter data center replication. It supports atomic write transactions and data partitioning across multiple servers and Data Centers. It was written in Erlang base on Riak Core (the core software piece of RIAK), and exposes a library of common objects based around the Conflict-free Replicated Data Type (CRDT). It relies on strong consistency between intra Data-center replicas and eventual consistency convergence between inter Data-center communication.

2.3 Peer-to-Peer

A peer-to-peer system consists in a distribution of tasks among peers (nodes) where tasks are dynamically allocated. A peer-to-peer system has a high degree of decentralization where peers implement both client and server funcionality to distribute bandwidth, computation, and storage across all the participants and few or none dedicated

peers exist in the system that own global state [26].

Once a peer is introduced into the system, there is little or no need for manual configuration. Peers are usually owned and operated by independent individuals who voluntarily join the system and are not controlled by a single organization.

Peer-to-peer also requires little or no infrastructure, usually the cost to deploy a peer-to-peer service is low compared to client-server systems. Peer-to-peer systems also exhibit an organic growth because the resources are contributed by participating nodes, meaning that a peer-to-peer system can grow almost arbitrarily without the need to upgrade the infrastructure, for example, replacing a server for a better one as is common practice when there is an increase in the number of users in a client-server system. This is because with each new node that joins, the systems increases in the amount of total available resources.

There is also the resilience to faults and attacks, since there are few (if any) nodes dedicated that are critical to the system's operation. To attack or shutdown a P2P system, there is the need to attack a large portion of nodes simultaneously, where with each new node joining the system an attack becomes harder to deploy.

Popular peer-to-peer applications include sharing and distributing files (like eDonkey or BitTorrent [40]), streaming media (like PPLive [37] or Cool Streaming [42]), telephony and volunteer computing.

2.3.1 Overlay Networks

An overlay is a logical network (typically defined at the application level) that abstracts the application from the physical network.

The network topology of the underlying (physical) network has a high impact on the performance of peer-to-peer services and applications. Therefore, it is essential to rely on an adequate overlay network for supporting systems in the right way. An overlay network is a logical network of nodes on top of the underlying physical network. It can be thought as a directed graph G = (N,E), where N is the set of participating nodes and E is a set of overlay links.

To achieve an efficient and robust delivery of data through a peer-to-peer system there is the need to construct an adequate overlay network. For this, the fundamental Architectural choices are the degree of centralization (partly decentralized vs decentralized) and the topology of the network overlay (structured vs unstructured).

Degree of centralization We can categorize Peer-to-Peer networks architectures by their use of centralized components.

Partly centralized overlay networks resort to some dedicated node or use a central server to perform some special control task such as indexing available resources or to provide a set of contact for nodes to join the system. New nodes can then join the overlay network by connecting to the controller. This overlay starts as a star-shapped because of the commnunication to the centralized unit by the participants and aditional overlay links are formed dinamically among participants that have been introduced to the controller. With this approach we get some of the downsides of a (single) centralized component, such as the existence of a single point of failure and potential bottleneck. Having this in mind, this system is not as reliable and resistent as a fully decentralized architecture. Still it provides organic growth and abundant resources, that are relatively simple to be managed via the single controller. Examples include Napster [22], Skype (an old version) [3], and BitTorrent using trackers.

Decentralized overlay networks do not use any form of dedicated nodes or centralized components that have a critical role in the operation of the system. In this type of overlay network, nodes that are joining the system are expected to obtain, through an outside channel, the network address of one of an already participating node in the system which serves as its entry point to the system. This makes the decentralized P2P more reliable compared to the partly decentralized system, avoiding the single point of failure and bottleneck issues, and increasing the potential for scalability and resilience. In this type of architecture some nodes may be used as super-nodes having increased responsibilities. A node becomes a super-node if it has a significant amount of resources, high availability, and a publicly routable IP address. Super-nodes can increase the efficiency of a P2P system, but may also increase its vulnerability to node failure.

Structured vs Unstructured Overlay: There is also the need of choosing between structured and unstructured overlays. This decision depends mostly on the usefulness of key-based routing algorithms and amount of churn (that is, when large numbers of peers are frequently joining and leaving the network at the same time) that the system is expected to be exposed to during operation.

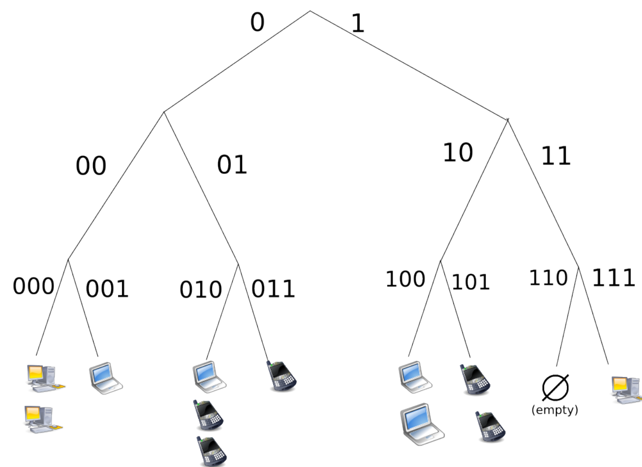


Figure 2.7: Structured P2P [39]

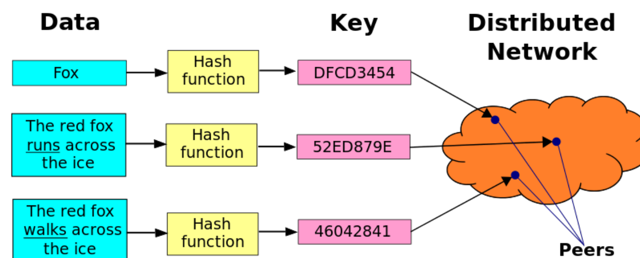


Figure 2.8: Representation of a Distributed Hash Table [40]

Structured overlays: In a structured overlay typically (Figure 2.7), each node gets an unique identifier in a large numeric key space, where the identifier will determine the position of the node in the overlay structure. Identifiers are chosen in a way that peers will be distributed uniformly at random over the key space. This allows to create a structure called DHT (Distributed Hash Table as depicted in Figure 2.8). This results in an easy lookup system that will work similar to an hash table, since if there is the need to communicate with a node there is only the need to know it's identifier (basically a key-value store). The issue that structured overlays bring it the trade-off, since the queries are much more efficient due to a faster lookup but it will cost in the performance when the churn is high because there will be the need to give identifiers to each new node that enters the system.

Unstructured overlays: In an unstructured overlay, (Figure 2.9) ,there is no particular topology formed by the network links and queries are usually done by flooding

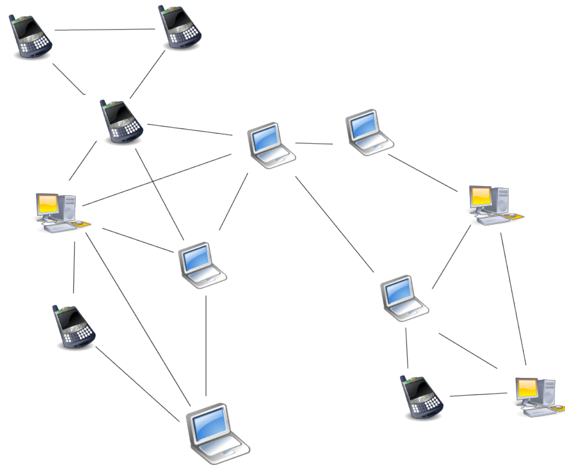


Figure 2.9: Unstructured peer-to-peer [39]

the network. Unstructured overlay networks are formed by establishing random links between the participants of the peer-to-peer system. In an unstructured overlay we get the opposite of what we get in a structured overlay, since we get less efficient queries having to flood the network to find as many peers as possible to locate a particular data object, resulting in the peers needing to process all the search queries, but in return we get a more robust system when churn is high.

2.3.2 Example peer-to-peer overlay networks

Chord [32] is distributed lookup protocol that was designed to enable peer-to-peer applications to efficiently locate the node that stores a particular data item. Chord provides support for just one operation: given a key, return the nodes responsible for the key. Keys are distributed over the nodes using consistent hashing and replicated over succeeding nodes. Nodes typically store their successor nodes, forming an ordered ring, making it easy to reason about the overlay structure. For fault-tolerance a list of successor nodes is kept and for efficient lookup a finger table, shortcuts to nodes over the graph, is used to perform large jumps in the ring topology.

2.3.3 Access Control Basic requirements on a P2P system

A peer-to-peer system is different from other systems where we are more used to see access control being applied. To apply access control on a peer-to-peer system some guarantees need to be made. In this case we will describe four main requirements that an access control model for P2P file-sharing networks should support [33]:

No centralized control or support: Traditional access control models, generally rely on central servers for authorization operations. This allows the existence of a single central location where the policies can be stored and evaluated. In a P2P network this doesn't happen, in fact, a peer has a significant level of autonomy and is in charge of storing and managing its own access control policies.

Encourage sharing files: One of the potentially desirable characteristics of P2P networks is the anonymity of the peers. Unlike client-server systems, peers in P2P systems are typically loosely coupled and provide very little information about their real-world identities. The interactions is done by peers that are mostly unknown. A P2P access control model must provide a mechanism for a host to classify users and assign each user different access rights, even if the users were previously unknown.

Limit spreading of malicious and harmful digital content: The open and unknown characteristics of P2P make it a good environment to malicious spreading and harmful content. A P2P access control system should support mechanisms to limit such malicious spreading and punish the ones responsible for it.

2.4 Geo-Replicated systems that use some sort of access control

In this section we will discuss some of the geo-replicated systems, also showing the ones that already implement some form of access control and how do they compare to the solution we propose in this dissertation.

It should be remembered that the goal of our solution is to provide access control systems with only weak consistency guarantees.

2.4.1 Data storage systems with eventual consistency

Although many protocols and investigation has been performed on data storage systems with eventual consistency, little emphasis has been given to the access control in this kind of systems.

We must recall that in a more organizational environment the option to choose systems with only eventual consistency is a very valid one being highly adopted because of the high availability and low cost that comes with it.

The original version of Amazon Dynamo [6] doesn't offer any type of authentication (subjects having to prove who they are) or authorization (checking if those same subjects are allowed to perform the desired operations).

Other data storage systems already offer some techniques for supporting access control but with low granularity, insufficient to provide control at the application level. Couchbase [5], MongoDB [23] and Riak KV [25] all support users, functions, and permissions, but with high granularity, at the level of buckets and data collections. Contrarily to these systems, our solution aims to be able to operate at the level of each object individually, without losing the benefits that arise from the use of weak consistency.

2.4.2 Systems without mutual trust

Wobber et al. [41] presented an access control for weakly consistent systems where there is no mutual trust between the replicas.

The scope is different since it consists in only a partial replication with different rights at each replica. This model also presents similar problems in relation to the causality between policies and subsequent operations that are allowed. In this model there is the existence of a waiting process for the desired policy, meaning that even with this the causality between changes of the policies that restrict the visibility of effects of operation and the subsequent execution of operation is not captured, which is something we aim at addressing. Because of this, and contrarily to our objective, this mechanism still allows the observation of information by other users when their observation rights were already revoked.

2.4.3 Systems with causal consistency

There are also some geo-replicated data storage systems that offer guarantees of causal consistency (mentioned in 2.2.1) as COPS [21] or ChainReaction [2] which have part of their motivation associated with an example similar to the one we will use on this dissertation. However, they assume that the list of access control is itself an independent object in the context of data storage, and consequently the propagation of causal operations between replicas located in distinct data centers avoid anomalies in which users can observe states to which they shouldn't have access.

The problem is that the cost to maintain the guarantees of causal consistency is not negligible, making it possible for the system to see it's performance deeply degraded. In contrast, our proposed solution doesn't require extra guarantees from the point of

view of the consistency model, avoiding in this way the additional costs that come with causal consistency.

2.5 Summary

This chapter discussed previous work in the areas related to the development of this dissertation.

In the peer-to-peer context the need for an overlay network has been described, explaining that different application requirements can require different types of overlays. Overlays can generally be described by degree of centralization and structured vs unstructured.

In the data-storage context we discussed about leveraging strong consistency with high availability. Different consistency models have been explored and, in the case of eventual consistency, several techniques for conflict resolution have been described.

In the collaborative-editing context, various commonly used approaches have been explored, describing how concurrency is handled in real-time editing in each of them.

In the next chapter it is explained the objective of our work and the model of the system.

Chapter 3

Theoretical Model

In this chapter we present a general perspective of the proposed solution.

There will be first a definition of the model of weak consistency and how the convergence is achieved (without access control) and after we give a formal model of the system including the access control where we start by introducing to some of the necessary nomenclature, the policies and the trust model, the semantics on how the model should behave in particular scenarios, the challenges and the techniques employed to solve these challenges.

In this Chapter there will be no discussion about implementation details, being this only a theoretical approach to the solution of the problem, Chapter 4 covers and discusses the implementation details with references to this chapter.

In more detail the following sections are organized as follows:

Section 3.1 presents how a model with weak consistency works and how convergence is achieved. It is also mentioned some assumptions that are made about the policies and trust model.

Section 3.2 presents how the model of access control should behave and what the users should observe at each time given a set of concurrent events in the system.

Section 3.3 in this section it is presented the challenges of introducing an access control to a system that follows the model of weak consistency. In this case it is discussed

only the challenges since only in chapter 4 are provided the ways to deal with them.

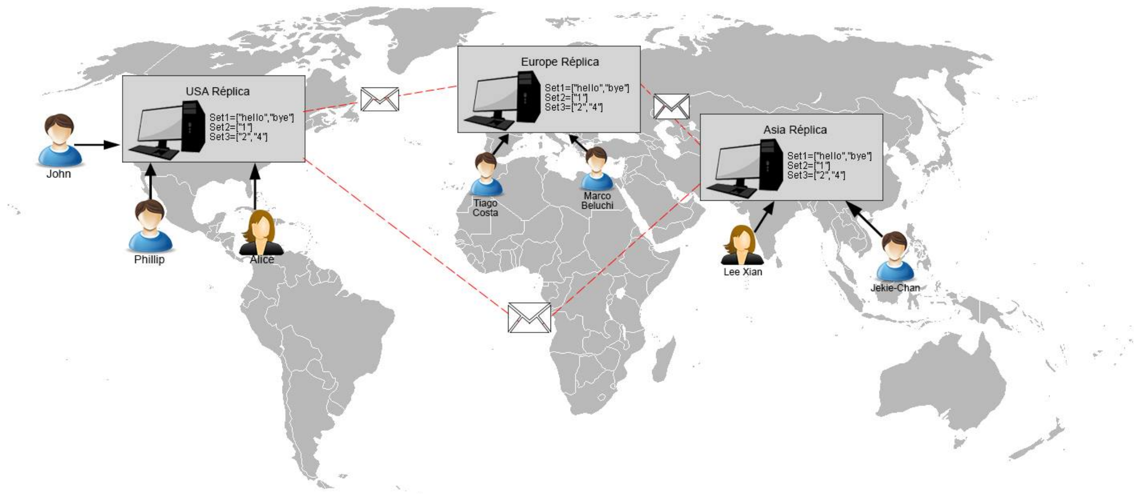


Figure 3.1: Geo-Replicated System Example

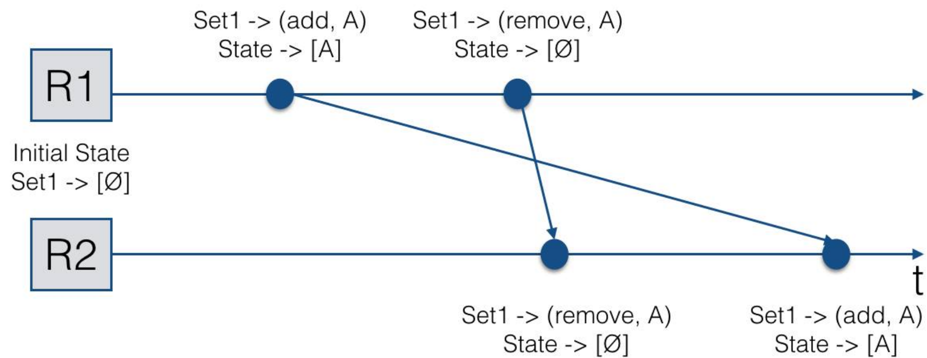


Figure 3.2: Inconsistency of data

3.1 Convergence on a Weakly Consistent Model

In this section it is presented a perspective on how a weakly consistent model works and how convergence is achieved. It will be presented the nomenclature and the backbone of a system that will be used to integrate the access control model.

Geo-replicated systems are used to provide lower latency and an increased availability to users. In those systems weak replication is used to deal with the CAP theorem [31] and to provide the system with the desired properties. When we have multiple servers the data should be replicated or partitioned between them all, in this case we will only discuss the replication of data. This means the data will be the same on all the servers so that users from all around the world can see the same content like shown in Figure 3.1, where the replicas distributed over USA, Europe and Asia, all have the same content, 3 sets with the same value in each one of them (i.e. we assume total replication).

In an initial state of the replication and without updates made on the data, all the servers will have the same state and the same representation of the objects, meaning we start the system from a converged state. Multiple servers exist so that different users can connect to each one of them to perform their actions, such as read or write operations. This means that each client will be connected to the server that has the lowest latency connection to itself, and clients will only perform the operations on that same server, as shown also in Figure 3.1, where users are connected to the replica that is closest to them, so that it is possible to achieve lower latency.

With each operation being made on each server independently, eventually there will be a state of divergence between the replicas, meaning that each of them will have a different state. This means that the updates need to be propagated between the servers, meaning that each one of them will receive the operations originally performed on the other replicas. The problem starts on the order by which each operation will be seen in a replica making it possible to exist a continuously state of divergence. A simple example using this image is an addition and removal to the set. If all replicas start with an empty set and one particular replica first receives an operation to include A in the set (add, A) and after this operation receives another operation to remove element A from the set (remove, A) that replica will have the empty set state at the end of performing both these operations. However it is possible that when those operations are propagated to reach the other replicas they do so in the opposite order, meaning that (remove, A) is executed first and only after (add, A) be received and executed, making the final state of such replica to include element A in the set, causing a state of continuous divergence, as shown in Figure 3.2.

Because of this, there is the need for resorting to mechanisms to solve this continuous divergence. In this case and for the rest of this dissertation we will focus on the use of CRDT's.

CRDTs are data types specifically implemented so that there is a convergence in

weakly consistent systems in the presence of concurrent operations guaranteed by design. There will be one CRDT at each replica and operations are propagated between them to guarantee that the different data replicas reach an (eventual) state of reconciliation, while also providing high availability.

With this weak consistency between replicas and with the use of CRDTs, we know that replicas may not return the same result at a given time, meaning that temporary divergence between replicas is possible, however, with the guarantee of convergence at soma unknown point in the future (i.e, eventually).

This is the system model in which our work was developed on and we have as an additional objective to solve all existing challenges without resorting to a more restrictive (i.e, stronger) consistency model.

3.1.1 System Model

This section discusses the system model in which the access control will be introduced. There will still be no notion of access control policies since that will be discussed in the next section (3.1.2).

To discuss the way a weakly consistent access control system should operate, we need first to look at the system model. The base model consists in a data management system that manages a set of objects, $Objs = \{o_1, o_2, \ldots\}$. The state of the object of the system can be modified by the execution of a set of operations, $Ops = \{op_1, op_2, \ldots\}$. For simplicity we will assume that an update changes the state of an object. The function $target(op)$ returns the object modified by the operation op.

Operations can be divided into read operations, Ops_{read} and write operations, Ops_{write}. Write operations are the only ones that may alter the state of the object, meaning that read operations won't change the state of the target object.

All operations are performed by a subject s. The tuple (op, s) is used to represent the execution of an operation op by the subject s. Operations can only be performed by subjects. The set of subjects of a system is designated by $Sujs = \{s_1, s_2, \ldots\}$.

Just like it is common in cloud systems, there is the assumption that the data management system keeps the replicas of all objects in multiple data centers. This way, for each distinct object, there is one (or more) replica in each data center. We assume a system operating only with guarantees of weak consistency and final convergence of data, as discussed in the section 3.1.

A user changes the state of an object by executing write operations in the replica to which he's connected by targeting an object with a given write operation, in which a

new state of the object o will be achieved by applying the desired operation op to the target object $target(op)$ by the subject (op, s). To this operation executed in the replica by the subject we call $UpstreamOperation$. Operations are asynchronously propagated in a reliable manner to other replicas, where they are executed locally on the (local copy) target object. This means that when a subject performs an operation on his local replica and in the object $o1$, that replica will then create a new operation that will be propagated to the other replicas so that they can execute the operation locally so that all converge to the same state. To this operation created on the initial replica we call $DownstreamOperation$. Only operations that modify the state of the objects are propagated to other replicas, meaning that Ops_{read} will not generate $DownstreamOperations$ on the replica where the operation was originally executed.

The state of an object at each replica, at any point in time is the result from the execution of $UpstreamOperations$ received directly from a user and of $DownstreamOperations$ receive from other replicas at that time.

Since we are referring to weakly consistent systems, we have to remind the reader that there is no guarantee on the order in which each operations will be executed. At the time of execution of an $UpstreamOperation$, multiple $DownstreamOperations$ are created and sent to the remaining replicas, but at the same time that same replicas may be also receiving $UpstreamOperations$ which makes it so that without any type of stronger consistency there will be no possibility of global ordering over the operations. In this case we assume that the system guarantees the final convergence of replicas, since all of them will execute the same set of operation and the storage resorts to the use of CRDTs (*conflict-free replicated data types*) [30], which as said in the previous section 3.1 it guarantees the final convergence of replicas.

Although there is no total order between operations, since we can't define the order of operations between replicas, we can still define a partial order (what happened before), which establishes a potential causality between operations. This is because in one replica we know the order by which the operations were executed. An operation op_a happened before op_b, $op_a \rightarrow op_b$ iff the effects of the operation op_a were known in the replica to which the user was connected when he executed the operation op_b which is exemplified in Figure **??**.

We can also say that two operations, op_a and op_b, are concurrent, $op_a \| op_b$ iff $op_a \nrightarrow op_b \wedge op_b \nrightarrow op_a$.

3.1.2 Policies and Trust Model

In this section it will be introduced the nomenclature of access control, followed by a brief discussion concerning the trust model assumed in this dissertation.

An access control policy defines the operations that each user can perform (at a given time). The attribution of rights to a certain subjects results in a triple $(r,s,o) \in Rights \times Subjs \times Objs$ which indicates that the subject s has the right r over the object o.

In this context, r represents the right to execute an operation op by $r \vdash op$. In our system, we define three different rights: the right of executing read operation, r_r where a user has the right to perform the operation of reading the value of an object, the right of executing a write operation, r_w where a user has the right to perform the operation of modifying the value of an object and we further introduce in our access control model a new type of operation consisting of the $Ops_{writeplus}$ which defines the the right of executing a writeplus operation, r_{wp} where a user can modify the access control policy of an object, with $\forall op \in Ops_{read}, r_r \vdash op$, $\forall op \in Ops_{write}, r_w \vdash op$ and $\forall op \in Ops_{writeplus}, r_w p \vdash_o p$.

A right (r,s,o) allows a subject s_1 to execute an operation op, $(r,s,o) \models (op,s_1)$ iff $r \vdash op \wedge target(op) = o \wedge s = s_1$.

The information relative to the access control policies, meaning, the triples with the attribution of rights, is maintained in the data storage system and updated in a similarly way to the data. This way, a user attributes or removes an access control rule by executing an operation in the replica to which he's connected, being that operation propagated asynchronously to all the remaining replicas so that all replicas can have the information required to verify the any request received for that particular object.

Another important topic to discuss when we are introducing an access control model on a distributed environment is the trust model. To define the trust model we need to assume the degree of confidence that the replicas have between them. In a simple geo-replicated environment we assume that all replicas belong to the same entity/company, and hence, we can assume that all replicas have a total trust between each other, meaning a mutually trustful system.

The assumption made in this dissertation, assumes a mutually trustful system and it is also assumed that the communication between replicas is safe and that the integrity of the messages exchanged between replicas is maintained. The security model we propose in this first part of the work, verifies the permissions of an update in the moment a subject makes an operation submission. This way, when a user contacts a

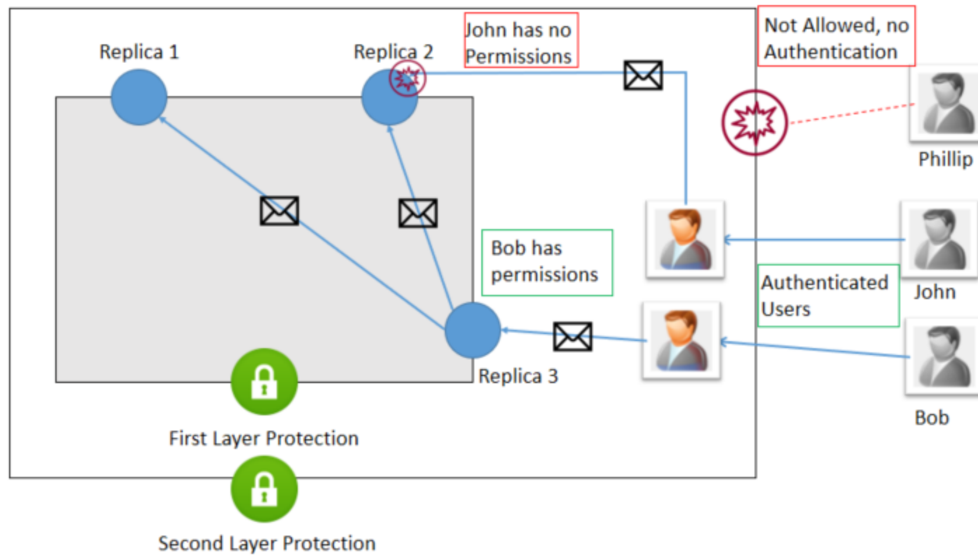


Figure 3.3: System trust

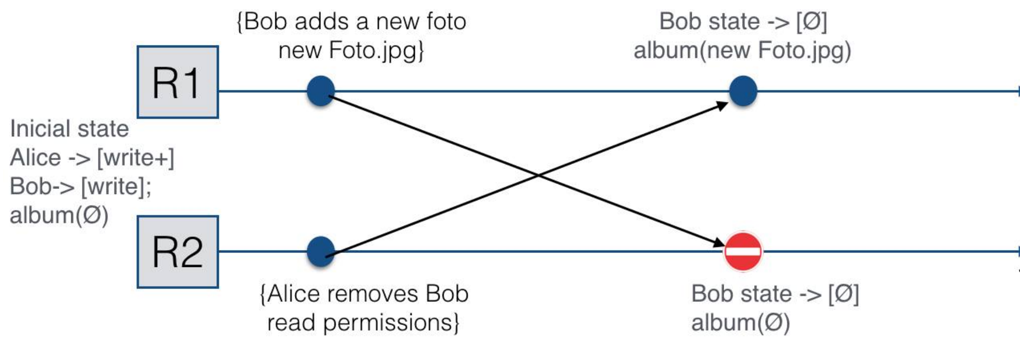


Figure 3.4: Possible data inconsistency

replica to execute an operation, that replica is responsible for verifying that the user has permissions to execute that operation, this means, the execution of $UpstreamOperations$ is verified but $DownstreamOperations$ are executed without the necessity to any kind of verification in relation to the actual policy since the verification was already made on the original replica and all the others trust the decision of the original replica.

This approximation is appropriate to a data storage system in the cloud, since there is the assumption that all the replicas of the system execute in a context of an organization that controls them, so it is not expected that they behave in a malicious manner.

This is only valid in systems where all the replicas trust each other and know that none of them may have malicious behavior (in a system where all the replicas belong to the same entity). The possibility of a malicious behavior would be for example, a replica to allow an update that is not allowed by the access control policy, which could lead all replicas to accept an update that shouldn't be allowed because of a single replica malicious behavior.

Figure 3.3 illustrates our system model, in which all replicas can communicate directly without the necessity to verify the permissions of the operations. We can assume that all the replicas are inside a type of black box, and each one of them will control the entry of operations in the black box by checking the locally know access control policy, but inside the black box they trade operations without the necessity to verify the policy. In more detail, a user authenticates itself in the system, the same will temporarily enter in the second layer of the protected environment. When he tries to perform an operation on the system ($UpstreamOperation$), that operation will be verified by the replica to which he's connected and if that operation is allowed it will enter the first layer of protected environment, where the operations are propagated without the need to make verification checks (this will only happen in Ops_{write} or $Ops_{writeplus}$ since Ops_{read} don't change the state of the system, only returning the local value of the target object). In the example of the figure, we have three different users that will try to join our system so they can perform operations. In this case Phillip may try to enter our system, but since he doesn't have an account, he won't be able to pass the authentication step, meaning he will stay out of the second layer of protection. John and Bob are both able to authenticate themselves, meaning that they enter in the second layer of protection. Each one of them will now try to perform one operation, John on the Replica 2 and Bob on the Replica 3. To perform the operations the replica's to which they are connected check their local access control policy, and Replica 2 responds by denying the operations to John since he has no permissions, not allowing that operation to enter in the First Layer Protection. On the other case, the operation performed by Bob is allowed, entering in the First Layer Protection and will be now freely propagated to all other replicas without the need to check the local access control policy again, so even if the permissions of Bob change, the operation will still be accepted in the other replicas because of the mutual trust that exists between the replicas (it is only required for one replica to check and accept the permissions). If this was not performed, a concurrent change to the access control policy rejecting Bob could leae the system in a permanently divergent state (as we discuss further ahead).

In summary, a unauthenticated subject will always stay out of the second layer of

the protected environment, meaning that it will be unable to perform operations on the system. Also a user that is authenticated may try to perform operations on the replica to which he's connected, but if he doesn't have the permissions to that operation they will be rejected.

This trust model allow us to surpass one of the challenges in relation to inconsistency of data and access control policies depicted in Figure 3.4. In this figure it is presented what occurs when there is a concurrent revocation of policies that belong to a user, and concurrently an operation changing the data by that user. As it is possible to see by this picture, the operation that will restrict the permissions will first be applied at replica R1 and will only be applied at replica R2 later. However, that data was already modified by Bob in replica R1, since when the operation was executed he still had permissions (locally), but when the operation reaches the replica R2 it will be rejected since concurrently Alice removed the writing permissions of Bob. This causes the system to continue in a constant state of divergence. With our trust model this issue disappears, since replica R2 will trust the decision performed by R1 and instead of verifying the local (and already modified) access control policy when the operation arrives, it will simply trust the judgment of replica R1 and accept the modification performed by Bob.

3.2 Access Control Semantics

For an access control system to be correct, the policies of access control should be followed by all replicas. In a strongly consistent system, the behavior of the system in one replica is represented by: an operation op performed by the user s is accepted, if when it was performed there was a right (r,s,o) that allows to execute it's operation, this is if $(r,s,o) \models (op,s)$. With this we know that for a strongly consistent system and for an operation to be accepted, there is the need to the previous execution of an operation to give the required rights to perform the operation, meaning that we need $(r,s,o) \rightarrow (op,s)$.

It is possible in a strongly consistent system to refer to an access control policy as the current, since it exists a total order between operations in the system. This way to know the current access control policy in a strongly consistent system, we only need to apply all the operations op made in the system by (a total) order since the initial state.

In a weakly consistent system it is possible to concurrently execute operations in different replicas. With the possibility of concurrent operations and the lack of total order among these operations, we can't assume the existence of a (single and global) current state of access control policy. Although we assume that there are guarantees of convergence of data to a final state in the presence of concurrent operations, there is the

need to decide which should be the behavior of the system in the presence of concurrent operations that modify the permissions of a subject and operations performed by that same subject. In the following we will consider the possible alternatives, considering separately all the relevant situations.

Read Operations: The read operations are executed only in the replica to which the user is connected. This property is fundamental to ensure a low latency in the execution of the operation. To maintain these properties, the only reasonable solution seems to be controlling the permissions in the replica in which the operation is executed based on what is currently locally known at that replica.

In the case where, concurrently, in another replica, an operation has been executed that changes the permissions of the subject, this operation will not have effects on the reading operation. For the concurrent operations to have effect, it would be necessary a coordination between all replicas before executing any operation, which would deny all the advantages provided by the use of weak consistency in particular, availability).

Besides, it is necessary to evaluate on how a system should behave when a read operation is allowed or denied in another replica:

Allow: When a subject s_1 is connected to a replica rp_1 and the replica acknowledges that the user doesn't have permissions to access an object o, all operations that the subject s_1 attempts to perform in that object should be denied. If another subject s_2 connected to another replica rp_2 with all permissions on the object o, wants to show him the new modifications on the object o (as for example a new photo the s_2 will introduce in it's album collection), that subject will perform the operation allowing the right of s_1 to read the object and make the write operation of adding the new photo. In this case the order to which the operations arrive at rp_1 will not be important since there's no possibility of information leakage, since the subject s_1 will simply be denied until the policy modification arrives at that replica.

Deny: In this case we have the opposite of what we discussed earlier in the Allow Read. We have one object o two subject s_1, s_2 and two replicas to which the subjects are directly connected respectively rp_1, rp_2. In this case we assume that s_1 has permissions to read the object o, and that s_2 have permissions to read and write on the object o. Assuming the addition of a photo in the album (object o) by the subject s_2 and that subject s_2 doesn't want s_1 to be able to

read the object after he added the new photo, subject s_2 will first perform the operation of changing the permission of s_1 on the object o so that he loses his read capabilities, and after he will add the new photo.

In this case a correct access control system, should never enable the subject s_1 to see the new photos added to the album unless the read capability of s_1 is reinstated by another operation. This way subject s_1 may still see the old content until the new policies arrived at his replica rp_1 but it should never be possible to him to see the new content of the object on the replica r_2 performed after the policy revocation of his read ability.

Write Operations: A write operation is executed in the replica to which the user is connected as an $UpstreamOperation$ and in all the other replicas as a $DownstreamOperation$. Contrary to reading operations, in this case, it would be possible to execute write operations and revert it's effects if at a later point it become know that a concurrent operation over the access control policy had revoked the write capability of the subject that performed the write. However, this would lead to the possibility of observing in the replica that processed such write the effect of that operation which would be later reverted, making it possible for an user to see an operation as being executed and allowed just for a few moments and later see the opposite. Due to this, in our system, write operations are only verified in the moment of their initial execution (this is, when they are executed as $UpstreamOperation$).

Besides it is necessary to evaluate how a system should behave when a write operation is allowed or denied in another replica concurrently:

Allow: This case is similar to Read Allow, as it was discussed earlier. Having two different replicas rp_1,rp_2 if an operation that permits write operations for a subject s_1 is performed in rp_2 and if s_1 is connected to rp_1, we know that temporarily it may be impossible for the subject s_1 to perform write operations, but eventually the update from rp_2 will arrive to rp_1 and it will be possible to s_1 to perform write operations.

Deny: In this case we will use again one object o which will consist in an album with photos, two subjects s_1,s_2 and two replicas to which the subjects are connected respectively rp_1,rp_2. In this case s_1 starts in both replicas with write permissions. We will assume that s_2 just wants to remove the write permission of s_1, and that afterwards s_1 will try to add a photo to o before his replica rp_1 receives the $DownstreamOperation$ issued by s_2 revoking his

write permissions. For this to happen s_2 performs the operation in the replica to which he's connected rp_2. This modification will only reach rp_1 (to which s_1 is connected) eventually. Due to this there is the possibility for rp_1 to allow the modification of the object o after there is an operation in rp_2 that revokes those permissions, being possible for s_1 to add a new photo to o.

In this case the way the system should behave is more application centered, since there is not a fully right and general answer. It is possible to assume that eventually we have to know that the operation of revocation happened earlier, and with this revoke the modification operation of s_1, with the object being an album we would see s_1 adding a photo and for s_1 that update would be accepted and after some time, it would disappear because of that revoke which can constitute an unexpected behavior to s_1. The other possibility would be always assume that if it is accepted in one replica, it should be accepted in all other replicas. In this case the new photo that s_1 added to o would be propagated and accepted on s_2. In our perspective the best way to deal with this problem is the second option and it's the one we will use on this dissertation, this because adding an additional photo will not force revocation of already accepted operations, and also because it would always be possible for s_2 to again delete this photo, since he keeps the permissions to edit the object o and it's policies. With this option we would have no rollbacks and with this a more consistent state of the system, only allowing for s_2 to see a new photo added by s_1 to which he had no permissions to add according to that state that s_2 sees in replica rp_2.

Both situations however are plausible, and one should choose the solution it should use according to it's system and application requirements. In our case we decided on the second option and propagate all operations once they are accepted in one replica. Although, this will create a new problem described next, enabling the opportunity for abuse by switching the replica to which the subject is connected.

Abuse by switching replica: With the possibility introduced in the last topic (Deny Write) of all updates accepted in one replica being propagated to all other without the permissions being verified again, so that the subject that introduces data doesn't suffer a rollback of her operations, we encounter another challenge.

It is possible for a subject to change the replica to which he's connected so

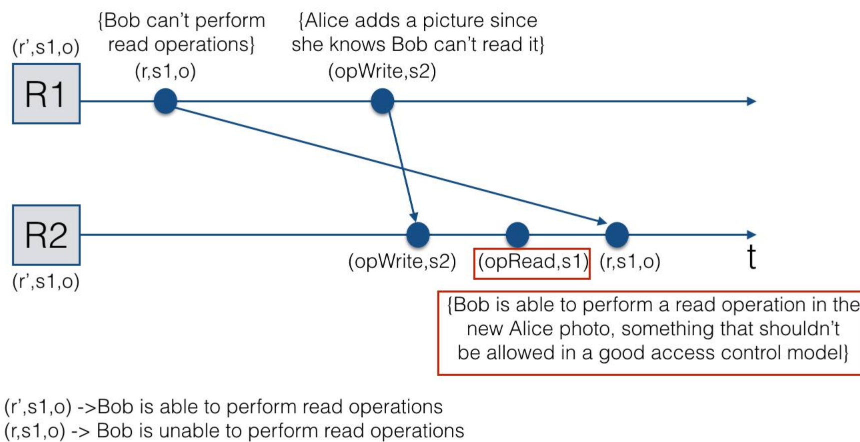


Figure 3.5: Violation of access control policies. Disordered operations.

that he can connect to a replica that still allows him to do certain operations. As we discussed in Deny Read there will be no problem in the read operation, but in the write it will be possible to a user to change for a replica in which he still can perform write operations to modify object o, even with him knowing that he should not have permissions. However, in this case we can assume the same we did on Deny Write. The subject to whom the permissions were taken will be able to modify the data while the propagation of operations doesn't arrive at his replicas, meaning that, if he is allowed to perform an operation at any given replica, he should be able to do it without any rollbacks, and the user that issued the operation to deny write operations can always issue write operations afterwards to ensure the desired final state of such object.

3.3 Inconsistency Challenges

In a system with access control, it is important to ensure the causality relations between the operations updating a access control policy and the ones updating the data to which the policy refers.

Considering the example in Figure 3.5, *Alice* tries to perform an operation on the replica to which she's connected to prevent the user *Bob* from accessing her photographs before she adds a new one. In a strongly consistent system, the order would be maintained and we would know that there would be a total order throughout all the system

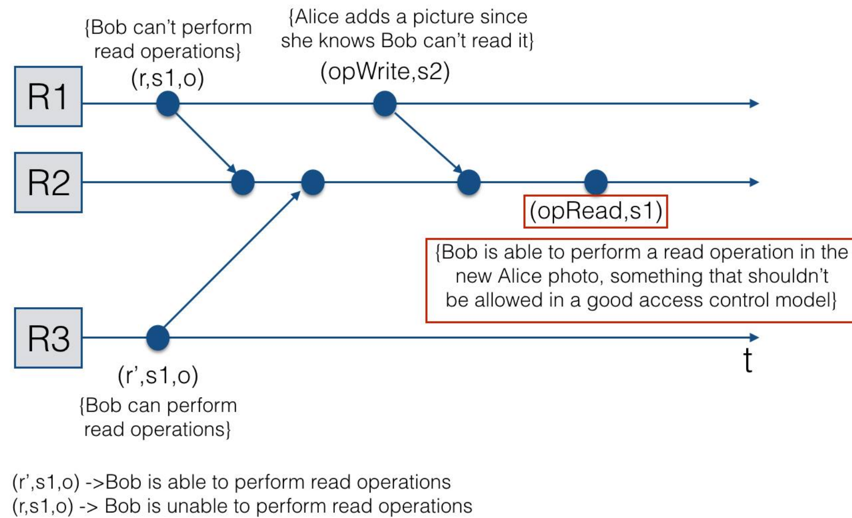


Figure 3.6: Violation of access control policies. Concurrent operations.

where $(r', s, o) \rightarrow (op, s)$, where r' are the new rights modified by Alice (subject s), on the list of her photos o, and the operation op would be the posting of the new photo by Alice. However in a weakly consistent system we don't have total order of operations. With this, if both operation are propagated independently and the system doesn't guarantee any type of causal consistency in the data access, it would be possible, in another replica, for *Bob* to see the new photo of Alice, just like is shown in the figure.

With this, it is fundamental to offer properties to ensure extra security measures so that it is not possible for information leakage when there are concurrent operations on objects and updates on the policies that restrict access to those same objects. Our system adds these new properties, as detailed in the next section.

3.3.1 Concurrent update on the access control policies

Besides it being possible to concurrently execute operations that read and update the data that modify user permissions, it is equally possible to concurrently execute operations that modify the permissions of the users. In our system we privilege a conservative approach to the most restrictive access control policies. This way, in the presence of two concurrent modifications to the access policy, after both are received in a replica, the one that will be active will be the most restrictive. To this restrictive property we call $RestrictiveMinimumPermissions$.

This approach allows avoiding the problems of information leakage represented in the Figure 3.6. In this example, Alice, after performing the operation to remove the read property to Bob, adds a new photo. If concurrently there was an operation allowing Bob to read Alice's album, with the *RestrictiveMiniumPermissions* Bob would still be unable to perform read operations on the album. However, without it, Bob would be able to see the photo and with this read a photo that Alice didn't wanted him to see, resulting in an information leakage.

This is a problem because of the knowledge the system should provide to users about the state of the system. When Alice wishes to add a photo that she doesn't want Bob to see, she will remove his permissions, making him unable to see the new photo. In a correct access control system, it should be possible for Phillip to also be able to change the permissions of Bob but only if he has the knowledge that Alice made the change. Without that knowledge he may not know the desire of Alice and make a wrong decision. A simple example would be if Phillip and Alice are two friends, but Phillip doesn't know that Alice argued with Bob, so he gives the permissions to Bob, but Alice argued with Bob, so she doesn't want her information to be available to Bob, and with this it is possible for Phillip to give the permissions and to go against the decision of Alice without his knowledge.

In short, and in a more formal notion, the execution of an operation op by subject s in the object o will be allowed iff $\exists(r,s,o) \in H : r \vdash op \wedge \nexists(r',s,o) \in H : ((r,s,o) \rightarrow (r',s,o) \vee (r,s,o) \| (r',s,o)) \wedge r' \nvdash op$, with H the set of operations know on the replica where the operation was initially executed.

3.4 Client to Client Communication Model

In the previous sections, we discussed the access control model in a system where the communication is done between centralized components that belong to the same entity, meaning that they have total trust between them allowing for secure communication between each other.

With the introduction of new models that allow the direct communication between clients (peers), taking an advantage of the increase in computational power in the client side, we decided to study how an access control could be applied to this model where clients communicate directly between them reducing the need for the centralized component. An example of this system is presented in Figure 3.7, where we have our centralized component with guarantees of secure access control, and the new direct communication between clients. We will discuss the best options to provide access

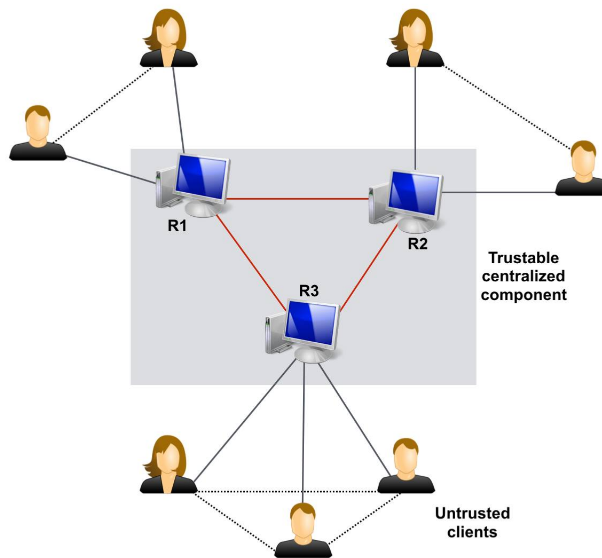


Figure 3.7: Direct Client Communication Model

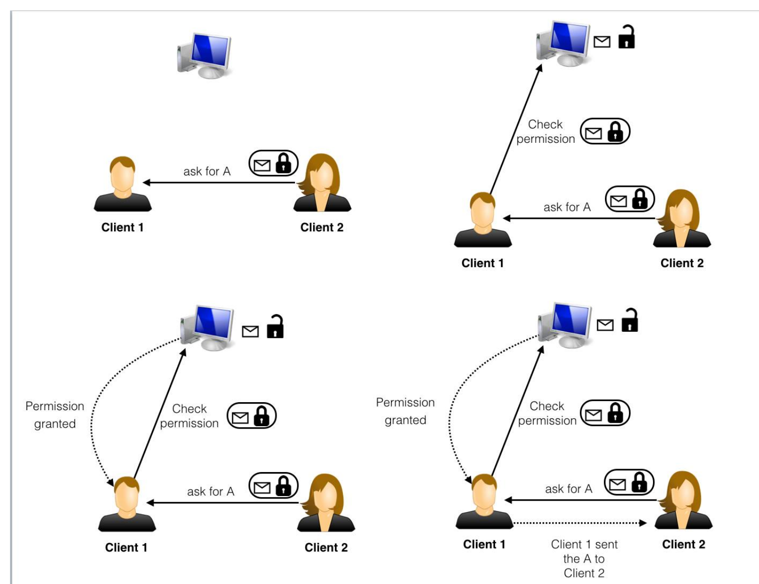


Figure 3.8: Client Communication with Centralized Component Request

control minimizing its impact on performance and necessity to communicate with the centralized component (which is the main focus of this model).

As shown in Figure 3.7 we have clients connected to the trusted centralized component but also to other clients they are close to, which they will try to get their requests from so there is no need to perform the request on the centralized component. This

follows the logic we used in the previous section, where users authenticate themselves in the system and the operations are verified in the centralized component node that received the operation and propagated to the rest. The difference is that clients also have a set of objects of the system, that they share among them.

The objective is to allow the users to share their objects but with restrictions, given that user1 may have an object from user2 that he doesn't wish to be shared with user3, and this brings the necessity of access control.

A simple solution would be for clients to maintain a version of the policies, so they can check it each time they receive a request from another client. This wouldn't work since each time there was an update over the access control policy, there would need to be a propagation of the new permissions between all the clients. To surpass this challenge the receiving client can request the state of the policies of the user he's receiving the operation from the centralized component. This removes the need to propagate the policies updates and it removes the possibility for a delay of an update in policies to allow information leakage. However this still causes the issue of doing too many requests to the centralized component, as shown in Figure 3.8. The user that receives the request still has to query the centralized component to check the permissions of the user that issued the request and the centralized should return (allow, disallow, not authenticated user), this makes that for each direct communication between clients there needs to be also a request to the centralized component, taking the away benefits of this model.

This happens due to the lack of trust between clients, where they can't trust each other to freely and correctly perform operations, and only the centralized components have knowledge of the policies, even if not in real time. Bringing the policies and the data together for the client (each object bind with it's policies) is as shown before, not a good solution, since client could themselves modify and propagate their local copies of the policy requiring clients to continually verify (reliable) policy information from the centralized component.

A different approach is also possible, that consists in following a strategy more similar to a capability based access control, where tokens are distributed to give access to the data. So if a user has the correct Token, he can make the changes to the object and propagate them, or read from another client. Also if a new user enters the system, there is only the need to provide him with the Token relative to the permissions he needs. This removes the necessity to keep a list of policies in the users and also removes the need to keep querying the centralized component each time a request is received by a client. Evidently such Token must be cryptographically protected by one central component, for instance by employing a cryptogenic signature using asymmetric cryptography.

However this approach has it's own downside, the removal of permissions is very costly and it may still allow for information leakage. The first problem is due to the fact that when there is a revocation of permissions, the Token for the object needs to change and the new Token needs to be distributed among (n-1) users, where n is the number of total users that have permissions for that object. The second problem is the safety of the subsequent data changes that the user performing the revocation may have, although this can be solved by the client checking in the centralized component if its permissions had been revoked.

In this case we didn't found a an adequate and fully general solution that will solve the issues related to a system enabling direct client communication, although if we are referring to make an implementation on a system that will have a low amount of revocation of permissions, than an approach closer to the capability-based access control is in our opinion the best solution.

3.5 Summary

This chapter describes the theoretical model proposed in the context of this thesis, showing what already exists and how it works and the challenges introduced by this model.

In a weakly consistent context, it was explained how this model works and how the state can vary. To get a basis for this work, it was also need to make some assumptions. This assumptions consisted in the premise that all the replicas of a system belong to the same company, meaning that arbitrary (i.e. byzantine) behavior will not exist on the replicas, ensuring that all of them will follow the enforcement mechanism correctly.

The next step was to explain the semantics of an access control, pointing the correct behavior of an access control system, explaining what should happen at all times given data access and (concurrent) changes in the access control policies.

Finally, we discusses alternatives to extend our model to architectures that are enriched with direct communication among clients, and where clients themselves own replicas of that objects. One example of such architecture is the legion system [20].

Chapter 4

Implementation

As it was discussed in the previous chapter, there were two different main challenges. The possibility of unauthorized access to information due to the independent propagation of updates on the access control policy and data, and also how to deal with multiple updates to the access control policy executed in a concurrent manner.

To test and integrate our solutions, we used the system Antidote[1], a data storage distributed system which uses the weakly consistency model with final convergence of data. In Antidote all data management resorts to the use of CRDTs.

For Antidote to be able to cope with access control policies that we wish to add, small implementations alterations had to be done, modifying it so that it is possible to use special versions of CRDTs that can encode their access control policy as well as enriching its interface so that operations are aware of the identification of the subject executing each operation.

While, for the implementation of our solution new CRDTs were developed which include specific information relative to the access control. This CRDTs maintain the generic properties of CRDTs, this is, they guarantee the final convergence of the state of replicas in the presence of concurrent updates.

In this chapter we will describe with more detail how the CRDTs on Antidote work and the existing operations, and the implementation of the new CRDTs that enable the use of an access control with properties that solve the challenges we discussed in the Chapter 3.

The following sections cover the aspects of the solutions and implementation:

Section 4.1 data objects existing in *Antidote* and how they work, and the though process behind the new structure of the CRDTs made for supporting access control.

Section 4.2 concurrent updates to our access control policies is one of the greater challenges faced in this work. In this section we will explain how the detection mechanism works so it is possible to detect concurrent updates and deal with them in the best way possible.

Section 4.3 concurrent updates of the access control policies may cause information leakage, for this we use *RestrictiveMinimumPermissions*, in this chapter we will explain in more detail how it works and it's implementation.

Section 4.4 the order to which operations are executed is one of the most important aspects in access control as discussed in the previous chapter. In this section we explain the implementation of the solutions to maintain a system that works correctly and without information leakage.

4.1 Data Object

To start developing our solution on Antidote there is the need first to know how the objects in the system work. Antidote already provides multiple operation based CRDTs, but for simplicity we will only focus on the pn-counter CRDT. Antidote uses operation-based CRDTs to achieve the convergence of data, meaning that these objects need to be able to perform the propagation of operations and be able to receive those same operations and achieve eventual consistency across all Replicas. Operation based CRDTs propagate operations to maintain the state in all replicas. In an operation based CRDT the operations propagated need to be commutative, while at the same time not idempotent being necessary Protocols that guarantee unique delivery, needed and included in Antidote.

A pn-counter CRDT consists of a positive-negative counter that supports both add and subtract operations. In a pn-counter CRDT we first start with the type of the object, which will consist of two different variables, one for the increments and other for the decrements. We can see an example of a pn-counter working with two different replicas, the *UpstreamOperations* applied are:

- R1 increments PN-counter, yielding a PN value of {1,0}

- R1 increments PN-counter, yielding a PN value of {2,0}
- R2 increments PN-counter, yielding a PN value of {1,0}
- R2 decrements PN-counter, yielding a PN value of {1,1}

The *DownstreamOperation* are then propagated with this values:

- R2 receives an increment notification from R1 and increments its P-counter from 1 to 2, yielding {2,1}
- R2 receives an increment notification from R1 and increments its P-counter from 2 to 3, yielding {3,1}
- R1 receives an increment notification from R2 and increments its P-counter from 2 to 3, yielding {3,0}
- R1 receives a decrement notification from R2 and increments its N-counter from 0 to 1, yielding {3,1}

Both R1 and R2 are now up to date with a value of the PN-counter {3,1}. The actual value of the PN-Counter is obtained by subtracting the N value (second element) from the P value (first element). So 3-1 is 2, which is the actual value of the PN-Counter.

With this we get a representation on how a general pn-counter CRDT works.

First there is the need to define the type of the data that will represent the CRDT, in this case, it consists of a representation of tuple that keeps the value of increments and decrements. The operation based CRDTs in Antidote consist of three main methods, as it is possible to see in other systems but with small variations on the method name. This three methods consist in $value$, $generate - downstream - operation$ and $update$:

- **Value:** It's the operation that will return the value of the CRDT. In this type of CRDT there is the need to return values so that read operation can be applied. This consist in *UpstreamOperations* that will return the final value of the local CRDT. With the pn-counter on Antidote, there are the positive and negative values, meaning that when a user makes a local read operation it is called the value method that will return the difference between the positive and negative variables on the object.

- **Generate-downstream-operation:** The $Generate-downstream-operation$ is the operation called when a subject tries to perform an update in one of the replicas. This operation creates the operations that are sent to all other replicas that belong to the system. In this method the object is not modified, being only created the operations needed to be sent for each replica including itself. This method receives an $UpstreamOperation$ consisting in a write operation, and creates $DownstreamOperations$ that are then propagated.

- **Update:** This method is responsible to execute the operations. It receives the needed information from the $Generate-downstrem-operation$, and uses that information to apply the operation modifying the CRDT state. This is the only operation that modifies the values of the CRDT. This operation is called whenever a replica receives a $DownstreamOperation$.

4.1.1 Applying Access Control to the Original Data Object

To implement our access control model, in our system, there was the need to develop new CRDTs. Our solution consists in adding to each CRDT it access control policies inside the object. This means that for example a pn-counter CRDT, will have it's basic data state with an positive and negative variables, but also a dictionary that will keep it's Access Control List.

The new CRDTs consists in a triple, containing $\{increment, decrement, policies\}$. The $policies$ consist of subject, rights, for example a subject Bob that may read and write will have an entry on the dictionary <Bob, [read,write]>. The possible permissions a subject may obtain are, **read**, **write** (make changes only to the object but not the policies), **writeplus** (ability to also make changes to the policy) and **own** (enables the possibility of deleting the object). The attribution of rights is sequential meaning that a user cannot have the permissions to perform a write operation without having permissions to execute the read operation.

In this new CRDT there was the need to make changes in the $Generate-downstream-operation$, $update$ and to add a new $value-policy$ operation. This changes consisted in:

- **Value-policy:** This method is responsible to return the current permissions value of a given subject. This operation is used when it is necessary to test if a given user has the permissions required to perform a given operation.

- **Generate-downstream-operation:** This operation is the one that is performed when $UpstreamOperations$ are executed over a replica. As discussed earlier, our

system makes the required permission verification only when it is issued by the user, meaning that it will be checked only on the method $Generate-downstream-operation$ when a replica receives an update. This operation will use the $value_{policy}$ with the identification of the subject that made the $UpstreamOperation$.

The method also modified to support modifications on the policies since this method needs not only to propagate operations about increments and decrements, but also about policy changes.

- **Update:** The update is responsible to change the object. In this case the changes made do update consisted in adding the possibility to receive $DownstreamOperations$ that changed the access control policies.

These methods are the backbone of the CRDTs. As they were described, they will not protect against the issues mentioned earlier, being the objects only able to store the data and change the policies.

In the next section it will be discussed the extra changes that had to be done to the CRDTs and to the Antidote system, so that the system can overcome the challenges mentioned in Chapter 3.

Also it is important to remember one of the bigger problems of the ACL (access control list), the problem with multiple users. If we have an application that would have an large number of users, we could have our CRDTs storing a lot of information since each user will have its own entry in our policies dictionary. To deal with this challenge, it is always possible to group the users into categories. This can be changed at the application level, considering that the subject that will submit the operation will consist in the identifier of a group. This makes it possible to reduce the number of entries in the access control policies.

Algorithm 1 CRDT pn-counter data type

1: $\{increment, decrement, policies\}$
2: $increment \leftarrow Integer$
3: $decrement \leftarrow Integer$
4: $policies \leftarrow Dictionary(\{subject, rights\}, IdentifierToken)$
5: $subject \leftarrow String$
6: $rights \leftarrow Setwithpossiblevalues[read, write, writeplus, own]$

Algorithm 2 GenerateDownstreamOperation pn-counter

Ensure: $Final_{Op}! = \{NoPermissions\} iff !policies.contains(s, r)$

1: **procedure** GENERATE-DOWNSTREAM-OPERATION(Op,s)
2:     **if** $op.equal(Increment \cup Decrement)$ **then**
3:         **if** $hasPermissions(write, s, policies)$ **then**
4:             Return $New_{DownstreamCountOp}$
5:         **else**
6:             Return $NoPermissions$
7:         **end if**
8:     **else**//$op.equal(Set - Right)$
9:         **if** $hasPermissions(writeplus, s, policies)$ **then**
10:             Return $New_{DownstreamRightOp} + policies$
11:         **else**
12:             Return $NoPermissions$
13:         **end if**
14:     **end if**
15: **end procedure**

Algorithm 3 Method to check if user has permission

1: **procedure** $hasPermissions$($right$,s,$policies$)
2:     $newList \leftarrow [read, write, writeplus, own]$
3:     $count \leftarrow 0$
4:     **for** each entry i in $policies$ **do**
5:         **if** $i.subject == s$ **then**
6:             $newList \leftarrow intersection(\text{i}.rights, newList)$
7:             $count++$
8:         **end if**
9:     **end for**
10:     **if** $count == 0$ **then**
11:         $returnfalse$
12:     **else**
13:         **if** $newList.isElement(right)$ **then**
14:             $returntrue$
15:         **else**
16:             $returnfalse$
17:         **end if**
18:     **end if**
19: **end procedure**

Algorithm 4 Update Operation Adding Right

```
1: procedure update(Op,s,currentPolicy)
2:     oldPolicy ← Op.policies
3:     right ← Op.right
4:     if oldPolicy == currentPolicy then
5:         policies.removeRights(s)
6:         policies.addRight(s, right)
7:     else
8:         policies.addRight(s, right)
9:     end if
10: end procedure
```

4.2 Concurrent Updates Detection

To deal with the challenges discussed in the previous chapter, there is the need to know how to detect the existence of a concurrent updates. As discussed in Chapter 3, it should be possible for multiple replicas to concurrently execute operations that modify the access control policies. Given this fact, it is possible that three different replicas may be changing the permissions of Bob at the same time. If there is no mechanism to preserve the state of the access control policy for Bob, in each replica two of those policy changes will be lost and with it possible important information and increases the possibility of information leakage.

Tackling this issue consists in detecting the existence of concurrent updates, and when they exist they should be dealt in a way so that all the policies on a subject are preserved. In a simple example, a subject Bob, which suffers two concurrent updates of his policies one giving read permissions and another giving read and write permissions, should have that information maintained in the system until a new change in policies occurs changing again his access control policies.

In this case, when there is a $Generate-Downstream-Operation$ that modifies the access control policies, part of the message propagated by that operation to all replicas is the current state of the ACL (P1). In each replica that receives the result of that $Generate-Downstream-Operation$ will perform the $Update$ operation. In the $Update$ operation it will also enter as argument the current value of the access control policies (P2). If the state of P1 is equal to the state of P2, that means that no concurrent operations were executed, and we can simply remove the old rights and add the new ones to that user, otherwise we should only add the new rights without adulterating the old ones as illustrated in Algorithm 5 (also in Appendix).

It should be noticed that in the pseudo-code it's compared the totality of the policies (current and old), but that's not what is effectively performed being only a simplification in the algorithm, since what should be compared is the policies of only that subject s.

4.3 Restrictive Minimum Permissions

Section 4.2, consisted in showing how to detect and maintain the complete information about the concurrent policy changes. This way there is no possibility for information to be lost, but there is still the challenge on how to choose the right policy when there are multiple current policies, so that we can completely protect our data.

In this case as discussed earlier, we will use the $RestrictiveMiniumPermission$. Having in account that section 4.2 leaves us with a system that under concurrent operations that modify the state of the access control policy, keep all the possible values, to have a secure system we may consider always choosing the option that will give us the best guarantee of security, meaning that we will only allow what is common on all the concurrent policies.

Illustrated in Algorithm 8 (also in Appendix), we show how this is achieved in our system. In this case, when there is the need to check if a user has the permissions to execute an operation, it will search in the policies for entries subject. If it encounters only one, there is only the need to compare it to the permission needed to execute the operation, otherwise if there are multiple, it is made the intersection between all the occurrences so that we can have the minimum value. If the user doesn't exist it will always return false.

With this, we get the guarantee that there's no possibility of leakage of information, when there's the existence of concurrent modifications of the access control policies, followed by an attempt to access that data object by a subject affected by that policy modification.

4.4 Operation Ordering Challenge

In this section we will describe the next challenge related to the ordering of operations (Figure 3.5) and how it was solved. To support this functionality, there were two possible distinct solutions, one that requires a system that guarantees the causality of operations over an object, and another that doesn't need causality guarantees but that uses more resources in the messages that are propagated between replicas.

In the next subsections it will be presented the two distinct solutions in more detail, subsection 4.4.1 containing the solution without causality and subsection 4.4.2 containing the solution with causality, knowing that the challenge consists in making sure the operations that add new data performed after the revoking of permissions, doesn't appear before in other replicas, allowing information leakage.

4.4.1 Without Causality

In this alternative, we have no guarantees that any operation that access data is only executed after any other operation of modification of policies that has occurred (logically) before.

To tackle this issue, this solution adds access control policies to all *DownstreamOperations*. When a write operation on data is performed, it will also propagate the current state of the access control policy of that replica. Using the example of the Figure 3.5, if Alice removes Bob reading permissions, and after it makes a write operation (for example, adding a photo), that write operation will be propagated (*DownstreamOperation*) together with information about the access control policies on the replica R1, meaning, that will contain explicitly the information that Bob cannot perform read operations. When that information reaches replica R2, the write operation performed by Alice will be performed at the same time that the access control policies are updated, since the write operation now brings additional information about the state of the policies in replica R1. This way, when Bob tries to perform a reading operation on R2, it will be calculated the minimum of his permissions as described in section 4.3, which will reflect the fact that Bob has lost his rights to perform reading operations, leading the request to be denied, maintaining the new information secure.

4.4.2 With the Use of Causality

The second option is more conventional, and instead of making more information go through the network, it simply delays the write operation so that it will only be applied after the access control policies modifications made on the original replica have also been applied.

This solution was achieved using the guarantees of object operation causality, meaning that each operation performed in a single object will have a version number (effectively by vector clock) and that all operations that modify an object are applied in sequence by each replica. Using again the example of Figure 3.5, if Alice removes Bob reading permissions, R1 will see that operation and will apply a version number to it

(in this case 1), and after it makes a write operation (adding a photo) which will have a new version number (in this case 2). If the operation marked with the version 1 arrives first, it will execute it immediately since it was the number of version R2 was expecting, otherwise if it receives the operation with version number 2, it will wait for the other operation, only executing the operation with version number 2 after he receives and executes the number 1. In this case we get the guarantee that Bob won't be able to see the new photo added by Alice, since his replica will wait for the new permissions, and only after that it will add the new photo to his replica.

It should be noticed that this is only possible because both access control policies and data belong to the same object, since Antidote only gives guarantees of causality per object.

4.4.3 Causality *vs* No Causality

In the two previous subsections it was described two distinct ways to deal with the problems of the lack of order provided by weakly consistent systems.

In short, the option of using causality should be chosen when we have a bigger guarantee that operations will not take too much time reaching the other replicas, meaning that even if the operations that are received are not ordered they will not take too much time being applied, without the necessity to add more information to each update operation message. This means the trade-off between the two solutions consists in the first without causality, adding additional information to each message which will put more pressure on the network but it once the message arrives the operation can be immediately processed, and on the second solution with causality, it will make some operations to take time to take effect even though they already reached a remote replica but they still need to wait for other operation from which they depend.

Also it should be considered the expected amount of time the system will run without the existence of possible disordered operations. In a highly unstable network the problem of the cost of causal consistency becomes even a bigger issue. If the system runs most of the time without any kind of disordered operations because the network allows it to, then the solution that uses causality will most likely be a better choice, since without problems in the network that cause disordered operations, the system will continue to apply the operations without waiting and it won't be needing the extra information on the messages that in bigger systems with a large amount of subjects may become non-negligible. Although as said earlier the application of causal consistency in systems is a costly process.

The option of our solution without causality also offers a higher level of protection. With causality described earlier we only ensure that for each replica to be aware that a given order operations issued by each individual object exists for each object, and that these operations must be applied in order. For example in Alice's case, if she performs two operations on the same replica, those operation need to be seen in all replicas by that order, meaning that if she removes the permissions to Bob and afterwards changes the state of the data of the CRDT, these two operations are executed by that relative order in all replicas.

However, if Bob already starts without reading properties, and concurrently Alice changes the data and someone else gives reading permissions to Bob, since those two operations are not causally related, there will be no protection of ordering in this case, meaning that even though Alice made the operation knowing Bob had no permissions, he will still be able to see the update because of the concurrent change of policy modification. In our solution without use of causality, that won't be the case. When the operation is sent to modify the data of the CRDT, the information about the permissions is sent with it, propagating the permissions that don't allow Bob to make read operations. Of course the system cannot protect that data from Bob for the rest of the time, since another subject with adequate permissions may give again the permissions to Bob to be able to make reading operations, but that already follows the expected semantics of a system with access control.

It should be also remembered that in current applications there is a strong rejection about using causal consistency because of it's associated overheads, meaning that a more general approach that can be applied to weak eventual consistent systems may be a better choice since it is more used in the real application use-cases.

With all this in mind, we show that it is possible to provide an access control under weak consistency without having to rely on any form of a stronger consistency. It was also discussed how causal consistency can also achieve this objective when we are referring to order of operations, although this was not our objective since our focus was to present a solution that can be integrated in a system with weak consistency which is the most used model.

4.5 Summary

In this chapter it is demonstrated the implementation adopted to tackle the issues described in the end of the previous chapter.

First of all there is a description on how the CRDTs work, and how the convergence of data is achieved. This is an important point of our work, since the use of special data types to achieve convergence is the building block of our solution. After, it is presented the discussion on how to apply access control to those data types.

The following topics in this chapter consist in the solutions given to solve the problems discussed in the end of Chapter 3, consisting in multiple methods to guarantee special secure guarantees and showing it's implementation and explaining their operation.

Chapter 5

Evaluation

In our solution we implement a system that doesn't restrict any of the properties of a weakly consistent system, only adding a small overhead in the propagation of operations that change data, making it also propagate the access control list. An evaluation about this overhead was also added, so it is possible to know how this mechanism affects the size of the messages used to propagate write operations.

In comparison to a system that uses strong consistency, our system will have a better performance since it avoids the burden of strong coordination among replicas and it is well known that a weakly consistent system has a better availability and lower latency than a strong consistency systems.

Also comparing to an hybrid system (data with weak consistency and policies with strong consistency), our system will follow the performance of the hybrid under a low amount of policy changes adding only some overhead because of the size of the messages transmitted, but will perform better under a high amount of policy changes.

In relation to systems that provide causality guarantees, it should be reminded the comparison between our system using causal guarantees and without (described in section 4.4.3). However the big difference between these systems is the cost of adding causal consistency, which because of this fact are less used in the industry, making our solution more fit to already existing solutions commonly found in real world deployments.

In the following sections the different evaluation and tests on our solution are presented. In this tests we only use one executing replica of the data storage system, and

simulate multiple other replicas that generate *DownstreamOperations* to this replica (the other multiple replicas are emulated using testing tools of the Antidote system). In this tests we use subjects that will be described in each test. This users are introduced directly in the operations without any need for authentication, since in this case we are focusing only in the correctness of the system, and not on the application in which it will be used. The permissions given to the users are incremental, being able to have permissions such as READ so that they can read the data, WRITE in which it is added the possibility to write in the data of the CRDT but not on his policies, WRITEPLUS that adds the capability of modifying the access control policies of the CRDT and OWN in which is added the possibility to delete the object.

These tests were performed using emulated multiple replicas that had the objective of producing *DownstreamOperations* to a single real replica, so it is possible to control it's state and see the results of the experience and document them and assess if it behaves as it should, following the semantics already described in this dissertation, in an environment in which we can actually force the existence of the problems described, such as multiple concurrent modifications to data and access control policies and the existence of disordered, data and access control policies, write operations.

The following sections are split into two different type of evaluations. The first consists in tests to evaluate the correctness of our solution given the issues discussed in this dissertation. The latter consists in an evaluation to measure the overhead introduced by those same solutions.

5.1 Correctness Tests

To properly evaluate our approach, the first step was to verify if the properties described in the previous sections were verified. In this section we validate our approach executing a set of operations, using them to perform correctness tests. These correctness tests will give the notion if the solutions previously presented, follows the discussed secure properties of how an access control system with weak consistency should behave.

The evaluation was performed by comparing our secure CRDTs described in this dissertation (with it's secure properties) and the regular CRDTs (with no access control), so that it is possible to compare the result both output in similar executions and which one follows the access control semantics defined in this dissertation. The CRDTs used in the evaluation consist in secure-pncounter-crdts. In both we compare the result also using assertions that force the systems to respect the desired state the system should have at each time and reporting if that doesn't occur.

The tests that were performed are split into three categories, under a sequential flow of operations where operations are seen in all replicas in the same order, under a sequential flow but where the operations reach another replica disordered and under concurrent operations.

The tables 'Operations' columns on this section consist as follows:

Replica Represents the replica where the operation is being executed and the identifier of the operation. If the replica name is followed by the number of an operation it means that same was an *UpstreamOperation*, otherwise it's a result of receiving an *DownstreamOperation* and updating it's value;

Subject Represents the Subject performing the operation. The possible values are the identifier of the subject or Update (if it consists in a *DownstreamOperation* update);

Operation Represents the operation being issued. The possible values are the operation or the update with the identifier of the original operation that created the *DownstreamOperation* received;

Object Value on the Replica Represents the local value of the object.

The tables 'Results' on this section consist as follow:

Operation The number of operation in the table 'Operations' in the column 1, in which the assertion was made;

Assertion The assertion that was made;

Secure CRDT result The result in our secure CRDT;

Normal Data Type Result The result in a normal CRDT.

This are the contents of the tables that will be present and discussed in the next subsections.

Table 5.1: Operations: Sequential Operations Test

Replica	Subject	Operation	Object Value on the Replica
1- R1 (Op1)	Alice	Increment: 3	{0,0,[{"Alice",[read,write,writeplus,own]}, {"Bob",[]}]}
2- R2	Update	Update de 1	{3,0,[{"Alice", [read,write,writeplus,own]}, {"Bob",[]}]}
3 - R1 (Op2)	Alice	Decrement: 8	{3,0,[{"Alice", [read,write,writeplus,own]}, {"Bob",[]}]}
4 - R2	Update	Update de 3	{3,8[{"Alice", [read,write,writeplus,own]}, {"Bob",[]}]}

Table 5.2: Result: Sequential Operations Result

Operation	Assertion	Secure CRDT result	Normal Data Type Result
2	AssertEqual value(Bob) == []	Ok	Ok
4	AssertEqual value_counter() == 3	Ok	Ok
6	AssertEqual value_counter() == -5	Ok	Ok

Table 5.3: Operations: Sequential Operations Test with Revocation

Replica	Subject	Opertation	Object Value on the Replica
1- R1 (Op1)	Alice	Increment: 5	{0,0,[{"Alice",[read,write,writeplus,own]}, {"Bob",[read,write]}]}
2- R2	Update	Update de 1	{5,0,[{"Alice", [read,write,writeplus,own]}, {"Bob",[read,write]}]}
3 - R1 (Op2)	Alice	Change Policy: {"Bob",[read]}	{5,0,[{"Alice", [read,write,writeplus,own]}, {"Bob",[read,write]}]}
4 - R2	Update	Update de 3	{5,0,[{"Alice", [read,write,writeplus,own]}, {"Bob",[read]}]}
5 - R2 (Op3)	Bob	Increment: 3	{5,0,[{"Alice", [read,write,writeplus,own]}, {"Bob",[read]}]}
6 - R1	Update	UpdateValue	{5,0,[{"Alice", [read,write,writeplus,own]}, {"Bob",[read]}]}

5.1.1 Sequential Operations

The first two tests (Table 5.1 and Table 5.3) and their respective assertion values (Table 5.2 and Table 5.4), show the results in a stable network, where all the operations are sequential and maintain their order.

The objective in this tests was to verify if our solution would perform correctly under a normal sequential execution.

In the first test, we start by performing an increment issued by Alice on replica R1. Afterwards Alice issues a new operation in this case consisting in an decrement. Both these operation were sequential, meaning that they were seen in the same order in R1 and R2.

As we can see in the results table Table 5.2, both CRDTs were able to deal with sequential operations, even without any assurance of special access control policies, which was expected since only in concurrent operations the normal data type should show incorrect behavior.

To be sure that sequential operations wouldn't create an incorrect behavior on the normal data type, we made again tests with sequential operations, but in this case we

Table 5.4: Results: Sequential Operations Test with Revocation

Operation	Assertion	Secure CRDT Result	Normal Data Type Result
2	AssertEqual value(Bob) == [read,write]	Ok	Ok
4	AssertEqual value(Bob) == [read]	Ok	Ok
6	AssertEqual value_counter() == 5	Ok	Ok
8	AssertEqual value(Bob) == [read]	Ok	Ok

Table 5.5: Operations: Disordered Operations

Replica	Subject	Operation	Object Value on the Replica
1-R1 (Op1)	Alice	Mudança Policica: {"Bob", []}	{0, 0, [{"Alice",,[read, write, writeplus, own]}, {"Bob", [read, write]}]}
2-R1	Update	Update de 1	{0, 0, [{"Alice",,[read, write, writeplus, own]}, {"Bob", []}]}
3-R1 (Op2)	Alice	Increment: 3	{0, 0, [{"Alice",,[read, write, writeplus, own]}, {"Bob", []}]}
4-R1	Update	Update de 3	{3, 0, [{"Alice",,[read, write, writeplus, own]}, {"Bob", []}]}
5-R2	Update	Update de 3	{3, 0, [{"Alice",,[read, write, writeplus, own]}, {"Bob", [???]}]}
6-R2 (Op3)	Bob	Read Operation on the Object	{3, 0, [{"Alice",,[read, write, writeplus, own]}, {"Bob", [???]}]}
7-R2	Update	Update de 1	{3, 0, [{"Alice",,[read, write, writeplus, own]}, {"Bob", [???]}]}

Table 5.6: Results: Disordered Operations

Operation	Asserton	Secure CRDT Result	Normal Data Type Result
5	AssertEqual value(Bob) == []	Ok	AssertEqualFailed - expected []; value(Bob) = [read,write]
6	AssertEqual value(Bob) == []	Ok	AssertEqualFailed - expected []; value(Bob) = [read,write]

revoked write properties of an user to verify if it would take effect. This can be seen in Table 5.3) where in operation Op2 Alice removes the permissions for "Bob" to perform write operations. As seen in the table results 5.4, both the data types worked as it should, meaning that it was confirmed that under sequential operations there was no need for additional security guarantees.

5.1.2 Disordered Operations

The previous tests showed that both solutions worked in an environment where the updates would be executed in a similar way. The next step in the correction tests was to force sequential operations to reach the other replica in a different order and evaluate if it allowed any data leakage in any of the data types. This test allowed us to verify the correctness of the system under a scenario where operations could reach another replica out of order, for example, Alice could perform a revocation operation and after an increment operation (locally), and the opposite being verified in another replica, which would leave the increment operation available for moments to the user to which Alice removed the permissions.

This test (Table 5.5 was performed using 2 replicas, as represented in Figure 3.5. The initial state of our object is {0,0, [{"Alice", [read,write,writeplus, own]}, {"Bob", [read, write]}]}. This means that in this object, we start with the following permissions. Alice can read the object, write on his data, write on the access control policies of the object and also delete the object. Bob in other hand, is only able to read the object and write on the data of the object.

In this case the objective is to verify if Bob can observe the new value of the CRDT by Alice after she had made the operation of removing the read capacity to Bob. As shown in the Figure 3.5, Alice executes its operations in replica R1 and Bob on the replica R2.

To test this we start by performing the operation of Alice removing the permissions of Bob ($Op1$) to make read operations which is executed locally and subsequent $DownstreamOperations$ are created to send to the other replica. In this case before we allowed that operation to reach replica R2 we made Alice perform a new operation consisting in increment the CRDT ($Op2$). To test if information leakage was possible we made it so that $Op2$ would reach replica R2 first than replica $Op1$, and between them Bob would perform a read operation $Op3$. The objective was to evaluate if Bob should be able to see the state of the object that had been incremented by Alice. As discussed in the previous chapters, a correct access control should not allow for Bob to see the new update of Alice, since as far as she knows she removed the permissions before submitting the new operation, believing that Bob wouldn't be able to observe it.

In the assertions Table 5.6, we can verify that there is a distinct result to this flow of execution in both data types. The assertion checks if at the time Bob performs the read operation he locally has the permissions to do it. In a normal data type he has the permissions, since the revocation of permissions only arrives at R2 after Bob has performed the read operation, allowing for leakage of information. However in our secure CRDT and due to policies being propagated with the data and afterwards the minimum of policies being calculated, we get the result consists in Bob being unable to read the object and with this, the new increment performed by Alice, performing correctly.

This is due to our secure system sending the permissions with the data, which means that when the Alice increment operation reaches R2, it already contains an entry that states that Bob has no permissions. When the evaluation to check if Bob has permissions enters in effect Bob has two different permissions read and empty, to which the restrictive minimum policy will return empty, not allowing Bob to perform the read operation.

Table 5.7: Operations: Concurrent Modification of Policies Test

Replica	Subject	Operation	Object Value on Replica
1-R1 (Op1)	Alice	Mudança politica: {"Bob", []}	{0, 0, [{“Alice”,,[read, write, writeplus, own]}, {“Bob”, [read, write]}]}
2-R1	Update	Update de 1	{0, 0, [{“Alice”,,[read, write, writeplus, own]}, {“Bob”, []}]}
3-R3 (Op2)	John	Mudança politica: {“Bob”, [read]}	{0, 0, [{“Alice”,,[read, write, writeplus, own]}, {“Bob”, [read, write]}]}
4-R3	Update	Update de 3	{0, 0, [{“Alice”,,[read, write, writeplus, own]}, {“Bob”, [read]}]}
5-R1	Alice	Incremento: 3	{0, 0, [{“Alice”, [read, write, writeplus, own]}, {“Bob”, []}]}.
6-R1	Update	Update de 5	{3, 0, [{“Alice”, [read, write, writeplus, own]}, {“Bob”, []}]}.
7-R2	Update	Update de 1	{0, 0, [{“Alice”,,[read, write, writeplus, own]}, {“Bob”, []}]}
8-R2	Update	Update de 3	{0, 0, [{“Alice”,,[read, write, writeplus, own]}, {“Bob”, [???]}]}
9-R2	Update	Update de 5	{3, 0, [{“Alice”, [read, write, writeplus, own]}, {“Bob”, [???]}]}.
10-R2 (Op3)	Bob	Leitura sobre objeto	{3, 0, [{“Alice”,,[read, write, writeplus, own]}, {“Bob”, [???]}]}

Table 5.8: Results: Concurrent Modification of Policies Test

Operation	Assertion	Secure CRDT Result	Normal Data Type Result
8	AssertEqual value(Bob) == []	Ok	AssertEqualFailed - expected []; value(Bob) = [read]
10	AssertEqual value(Bob) == []	Ok	AssertEqualFailed - expected []; value(Bob) = [read]

5.1.3 Concurrent Operations

The previous tests showed that both solutions worked in an environment where the updates would be executed in a similar way, and that only our solution would protect leakage of data when operations arrive to other replicas in an disordered manner. The next step in the correction tests was to force concurrent modifications of policy rights to verify which would be the final result. This test allowed us to verify the correctness of the system under a scenario where two different principals would perform operations to change the permissions of another principal and evaluate if it was possible to exist leakage of information.

The initial state of the target object is {0,0, [{"Alice", [read,write,writeplus,own]}, {"Bob", [read,write]}, {"John", [read,write,writeplus,own]}.

The operations that were performed are described in Table 5.7, following the same structure as the one presented before in the Table 5.5. To test this we resort to the example illustrated in Figure 3.6 in which exists a concurrent modification of the access control policies of Bob (replica R1 and R3). In this example, Alice performs an increment operation after removing the read permissions Bob previously had in R1, and another user John in replica R3 performs concurrently to the operations of Alice, an operation that allows Bob to perform reading operations. After that Bob tries to observe the state of the object in R2. In this case the operation of removing the permissions of Bob performed by Alice is the first operation reaching R2, afterwards the operation adding

permissions to Bob performed by John arrives to replica R2 and only in the end the increment operation performed by Alice will reach the replica R2 which is followed by a read performed by Bob.

In this case the main concern is the value present in the replica R2 after the execution of the sixth operation (when the increment arrives to replica R2). As previously discussed, the expected the result in a correct access control system should be the deny of permissions on his read operation, since Alice removes the permissions of Bob and makes the increment knowing that Bob shouldn't be able to see her modifications.

The Table 5.8 shows the assertions used to test this case and the results achieved using our secure solution and native CRDTs that don't provide our secure properties. The table structure follows that one already described for Table 5.6. In this case the assertions consisted mainly in verifying the state of the read permissions of Bob when he tried to perform the read Operation ($Op3$). In a normal data type, Bob has the read permissions, this happens due to the update performed by John being the last to arrive to R2, which in a normal data type will be the value that will stay in the object. However, the same is not verified using our secure CRDTs. This happens due to the detection of concurrent updates and also to the use of a restrictive minimum policy. When the update of policy from John reaches the replica R2, it realizes that John and Alice operations were performed without knowing the other intention, and with this it will keep both policy values in store. Afterwards when Bob tries to perform a read operation the minimum value of policy will be calculated and the more restrictive permissions of Alice will win, not allowing Bob to perform the read, respecting the correct behavior of an access control system.

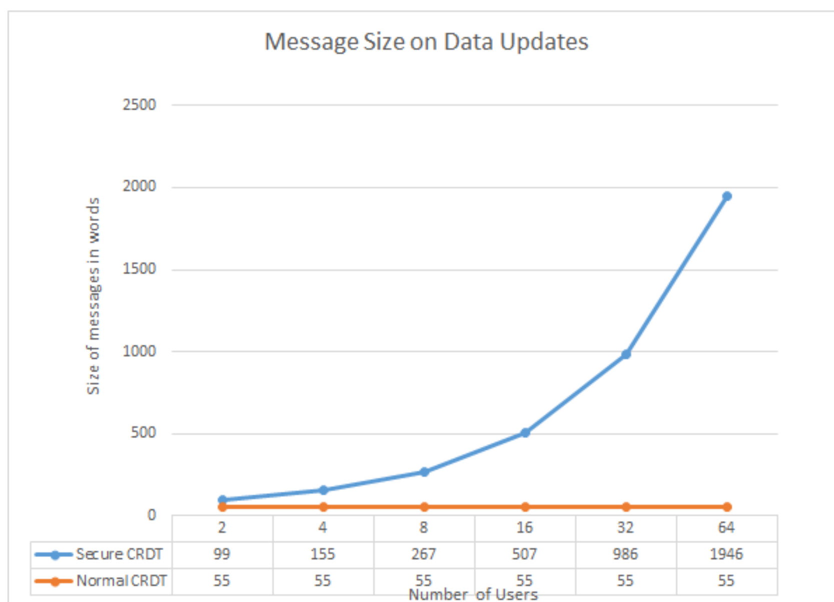
Has shown in Tabels 5.6 and 5.8 and its tests, by using our solution, the system always behave correctly, following the semantics of how a system with access control should behave, never allowing the leakage of information of the modifications performed by Alice. Systems that only use native CRDTs and with no extra security guarantees, on contrary, show their weaknesses for example in this case, by allowing Bob to read updates that he shouldn't be allowed to, clearly demonstrating the dangers of privacy inherent to data storage systems that operate under the model of weak consistency.

5.2 Overhead

To give a better overview of our system we also conducted evaluation to measure the overhead introduced by our solution, consisting in tests to check the amount of time

operations took to execute and the size of the messages exchanged between replicas.

To test the amount of overhead introduced to the update operations in terms of execution time, we measured the average of the execution time of an update function consisting in the increment by four of the CRDT. In this case the measurement was made using microseconds, where on average and median, the time it took to complete the operations was close to 0 mics, having only a peak max value of 12 mics. In the same test for the generate downstream operation, the results were an average of 2 mics and median 1 mics. In both this tests, the multiple operations to be able to give us a median and average and a good comparing state, were performed in 50000 mics. Doing the same with a CRDT that doesn't provide our secure properties also gives us the result of an average 2 mics and median 1 mic, showing negligible overhead to our solution. Comparing to a solution without access control policies (meaning the use of a simple CRDT that doesn't provide access control restrictions) the result of generate downstream operation changes to a median of 1 mics and an average of also 1 mics, showing that the overhead on the execution level is not too high.



	2	4	8	16	32	64
Secure CRDT	99	155	267	507	986	1946
Normal CRDT	55	55	55	55	55	55

Figure 5.1: Message Size on Data Updates

To evaluate the difference of the message size, two different charts were made, the first giving the message size of data operations (in a counter the increment and decrement operations) and the latter giving the message size of policy operations. On both

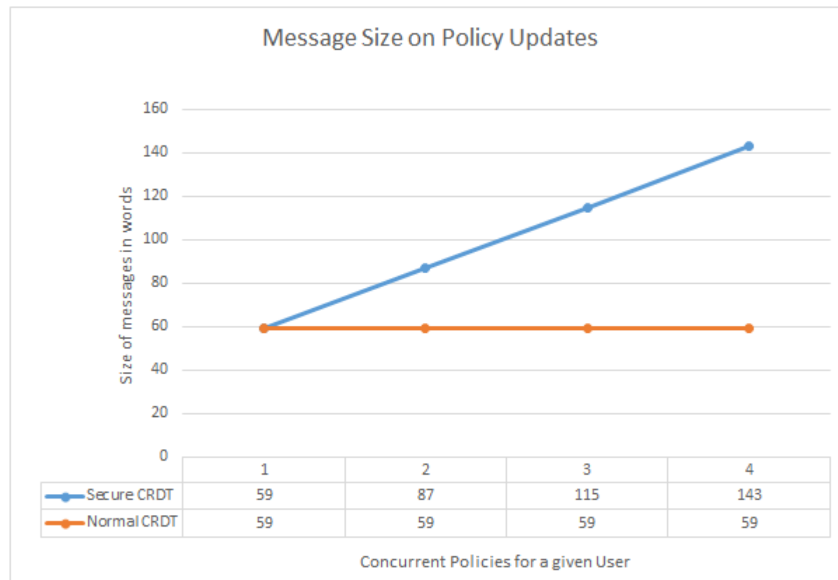


Figure 5.2: Message Size on Policy Updates

evaluations the Y-axis represents the result of the operations erts debug:size, that represent the words needed in erlang.

The first set of results is represented in the Figure 5.1. As mentioned on this thesis, the way to prevent unauthorized access to data was to propagate the information on the policies at the time the operation was done on the replica in which the request originated. This makes the size of the message propagated between replicas to increase with the amount of users existing in the system. As shown in the results in the Figure 5.1 an increase of the number of users drastically increases the size of messages. In systems that have a large amount of users, the system should use group names instead of user names, becoming a system closer related to a Role Based Access Control, that will reduce the message size drastically. To this test we used erlang command erts debug:size to measure the size of the operations propagated in words.

This is verified due to the fact that our CRDTs propagate the current policies when performing an operation that modifies the state of the data (not the permissions).

This means that our solution produces an high overhead that increases with the number of new users that join the system making our solution viable only if we take advantage of a model closer to a Role Based Access Control where users are grouped, meaning less identifiers and less message size.

The second evaluation is represented in the Figure 5.2. In this case we evaluate the size of the messages when a policy change is made (when there is a set right operation).

In this case we only propagate with the operation the state of the rights of the user we are changing the rights. This means that there are only four different possible values. In the Figure 5.2 it is represented the size of messages when there is a propagation of a change of policy rights on a user in which before had been target of n concurrent policy operations, where n is captured in the X axis (1 to 4).

This means our solution will provide a negligible overhead to operations that change the access control policies, which in addition to the time tests performed, make it a good solution in systems that have a large amount of change of policies operations. This however goes against what is verified in 5.1, since usually a system with a large number of policy updates will only happen in a system that has a large number of users, which makes data operations costly in message size.

5.3 Summary

In this section we were able to prove with correctness tests that our solution could perform so that no leakage of information was possible, and with this fully protect the users from situations where it's sensible information could be temporarily available to unauthorized users.

Also the evaluations performed to verify the overhead introduced were conclusive in a way that showed our solution only introduced a reasonable overhead to the message size of operations that modify the data (not the policies). Although as discussed this can be attenuated by using a model closer to Role-Based Access Control, where each user is classified into a group and only those groups identifiers have permissions.

In Summary our solution offers the desirable security guarantees while adding only a partial overhead to what precious already existed (and that does not provide the desirable security guarantees).

Chapter 6

Conclusion

In this dissertation we tackled the challenges associated with access control in the context of data storage systems that offer weak consistency. This systems are extremely relevant due to the proliferation of geo-replicated systems that support most of the large scale systems nowadays. Our main objective was to produce a way to provide access control in the most correct way possible and supporting properties that would make an access control to be viable in the real world scenarios, without the necessity to provide stronger consistency properties which are both more costly and less used in the industry.

Contrarily to other alternatives, we don't need guarantees of causal propagation or stronger forms of consistency, since we integrate the policies directly at the object level in the data storage system, also giving enough granularity to be able to control the access to each specific object. Our evaluation shows that our approach, integrated in the Antidote system, provides the desired properties for a mechanism of access control. The evaluation was more centered on verifying the correctness of the system, since it is already know that stronger consistency properties make the system less available, and the goal of our work was to make sure that even with the addition of access control policies we wouldn't disrupt the advantages of using a system with weak consistency such as the improved availability and lower latency of operations.

As it was mentioned in the topic of access control on a peer-to-peer environment, we can in the future extend this mechanism of access control to be able to achieve a correct

(and trustful) execution in environments where the replication of data is extended as to include the clients, with capacity of propagating operations between them. In this dissertation we already give an introduction to this systems and discuss alternatives to solve the issues they bring, however the implementation and evaluation in this scenario will be addressed as future work.

An example of a system that behaves as described is the Legion system [20], which tries to enrich the form of how the web applications are constructed, offering a direct interaction between clients to increase the availability and lower the latency in the propagation of operations, creating a *peer-to-peer* network between clients in a transparent fashion.

Publications:

Part of the results in this dissertation were published in the following publications:
Controlo de Acessos em Sistemas com Consistência Fraca Tiago Costa, Albert Linde, Nuno Preguiça e João Leitão. Actas do oitavo Simpósio de Informática, Lisboa, Portugal, September, 2016.

Bibliography

[1] 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: *Proc. 36th IEEE International Conference on Distributed Computing Systems (ICDCS 2016)*. June 2016.

[2] S. Almeida, J. Leitão, and L. Rodrigues. "ChainReaction: A Causal+ Consistent Datastore Based on Chain Replication". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys '13. 2013, pp. 85–98.

[3] S. A. Baset and H. Schulzrinne. "An analysis of the skype peer-to-peer internet telephony protocol". In: *arXiv preprint cs/0412017* (2004).

[4] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. "Spanner: Google's globally distributed database". In: *ACM Transactions on Computer Systems (TOCS)* 31.3 (2013), p. 8.

[5] Couchbase. *Couchbase*. URL: http://www.couchbase.com/.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. "Dynamo: amazon's highly available key-value store". In: *ACM SIGOPS Operating Systems Review*. Vol. 41. 6. ACM. 2007, pp. 205–220.

[7] Douwiki. *ACL*. URL: https://www.dokuwiki.org/acl.

[8] D. Ferraiolo, D. R. Kuhn, and R. Chandramouli. *Role-based access control*. Artech House, 2003.

[9] M. Frank. *What is RBAC?* URL: http://www.mariofrank.net/rolemining.html.

[10] N. Hayashibara, X. Defago, R. Yared, and T. Katayama. "The φ accrual failure detector". In: *Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on*. IEEE. 2004, pp. 66–78.

[11] V. C. Hu, D. Ferraiolo, R. Kuhn, A. R. Friedman, A. J. Lang, M. M. Cogdell, A. Schnitzer, K. Sandlin, R. Miller, K. Scarfone, et al. "Guide to attribute based access control (ABAC) definition and considerations (draft)". In: *NIST Special Publication* 800 (2013), p. 162.

[12] V. C. Hu, D. R. Kuhn, and D. F. Ferraiolo. "Attribute-based access control". In: *Computer* 2 (2015), pp. 85–88.

[13] A. Imine, A. Cherif, and M. Rusinowitch. "A flexible access control model for distributed collaborative editors". In: *Workshop on Secure Data Management*. Springer. 2009, pp. 89–106.

[14] X. Jin, R. Sandhu, and R. Krishnan. "RABAC: role-centric attribute-based access control". In: *Computer Network Security*. Springer, 2012, pp. 84–96.

[15] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web". In: *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. ACM. 1997, pp. 654–663.

[16] R. Klophaus. "Riak core: building distributed applications without shared state". In: *ACM SIGPLAN Commercial Users of Functional Programming*. ACM. 2010, p. 14.

[17] R. Kohavi and R. Longbotham. "Online experiments: Lessons learned". In: *Computer* 40.9 (2007), pp. 103–105.

[18] A. Lakshman and P. Malik. "Cassandra: a decentralized structured storage system". In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40.

[19] J. Li. *Distributed Systems - Lec 12: Consistency Models – Sequential, Causal, and Eventual Consistency*. URL: http://www.cs.columbia.edu/~roxana/teaching/DistributedSystemsF12/lectures/lec12.pdf.

[20] A. Linde. "Enriching Web Applications with Browser-to-Browser Communication". MA thesis. Faculdade Ciência e Tecnologia, Universidade Nova de Lisboa, July 2015.

[21] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. "Don'T Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS". In: *Proc. of the 23rd ACM Symposium on Operating Systems Principles*. SOSP '11. Cascais, Portugal, 2011, pp. 401–416. ISBN: 978-1-4503-0977-6.

[22] L.Naspter. *Napster*. URL: http://www.napster.com.

[23] MongoDB. *MongoDB*. URL: https://www.mongodb.com/.

[24] G. Oster, P. Urso, P. Molli, and A. Imine. "Proving correctness of transformation functions in collaborative editing systems". In: (2005).

[25] Riak. *Riak KV*. URL: http://basho.com/products/riak-kv/.

[26] R. Rodrigues and P. Druschel. "Peer-to-peer systems". In: *Communications of the ACM* 53.10 (2010), pp. 72–82.

[27] P. Samarati and S. D. C. Di Vimercati. "Access control: Policies, models, and mechanisms". In: *Lecture notes in computer science* (2001), pp. 137–196.

[28] R. S. Sandhu. "Role-based access control". In: *Advances in computers* 46 (1998), pp. 237–286.

[29] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. "A comprehensive study of convergent and commutative replicated data types". PhD thesis. Inria–Centre Paris-Rocquencourt, 2011.

[30] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. "Conflict-free replicated data types". In: *Stabilization, Safety, and Security of Distributed Systems*. Springer, 2011, pp. 386–400.

[31] S. Simon. "Brewer's CAP Theorem". In: ().

[32] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. "Chord: A scalable peer-to-peer lookup service for internet applications". In: *ACM SIGCOMM Computer Communication Review* 31.4 (2001), pp. 149–160.

[33] H. Tran, M. Hitchens, V. Varadharajan, and P. Watters. "A trust based access control framework for P2P file-sharing systems". In: *System Sciences, 2005. HICSS'05. Proceedings of the 38th Annual Hawaii International Conference on*. IEEE. 2005, pp. 302c–302c.

[34] S. University. *Capability-Based Access Control*. URL: http://www.cis.syr.edu/~wedu/Teaching/CompSec/LectureNotes_New/Capability.pdf.

[35] R. Van Renesse, D. Dumitriu, V. Gough, and C. Thomas. "Efficient reconciliation and flow control for anti-entropy protocols". In: *proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*. ACM. 2008, p. 6.

[36] W. Vogels. "Eventually consistent". In: *Communications of the ACM* 52.1 (2009), pp. 40–44.

[37] L. Vu, I. Gupta, J. Liang, and K. Nahrstedt. *Insights into PPLive: A Measurement Study of a Large-Scale P2P IPTV System*. Tech. rep. Technical report, 2005.

[38] M. Weber, A. Bieniusa, and A. Poetzsch-Heffter. “Access Control for Weakly Consistent Cloud-Storage Systems”. Submitted for publication. 2016.

[39] Wikipedia. *Bittorrent(protocol)*. URL: http://en.wikipedia.org/wiki/BitTorrent_(protocol)#Adoption.

[40] Wikipedia. *Peer-to-peer*. URL: http://en.wikipedia.org/wiki/Peer-to-peer.

[41] 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*. ACM. 2010, pp. 293–306.

[42] X. Zhang, J. Liu, B. Li, and T.-S. P. Yum. “CoolStreaming/DONet: a data-driven overlay network for peer-to-peer live media streaming”. In: *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*. Vol. 3. IEEE. 2005, pp. 2102–2111.

Appendix

Algorithm 5 Update Operation Adding Right

1: **procedure** $update(Op,s,currentPolicy)$
2: $oldPolicy \leftarrow Op.policies$
3: $right \leftarrow Op.right$
4: **if** $oldPolicy == currentPolicy$ **then**
5: $policies.removeRights(s)$
6: $policies.addRight(s, right)$
7: **else**
8: $policies.addRight(s, right)$
9: **end if**
10: **end procedure**

Algorithm 6 CRDT pn-counter data type

1: $\{increment, decrement, policies\}$
2: $increment \leftarrow Integer$
3: $decrement \leftarrow Integer$
4: $policies \leftarrow Dictionary(\{subject, rights\}, IdentifierToken)$
5: $subject \leftarrow String$
6: $rights \leftarrow Setwithpossiblevalues[read, write, writeplus, own]$

Algorithm 7 GenerateDownstreamOperation pn-counter

Ensure: $Final_{Op}! = \{NoPermissions\} iff !policies.contains(s, r)$
1: **procedure** GENERATE-DOWNSTREAM-OPERATION(Op,s)
2: **if** $op.equal(Increment \cup Decrement)$ **then**
3: **if** $hasPermissions(write, s, policies)$ **then**
4: Return $New_{DownstreamCountOp}$
5: **else**
6: Return $NoPermissions$
7: **end if**
8: **else**//$op.equal(Set - Right)$
9: **if** $hasPermissions(writeplus, s, policies)$ **then**
10: Return $New_{DownstreamRightOp} + policies$
11: **else**
12: Return $NoPermissions$
13: **end if**
14: **end if**
15: **end procedure**

Algorithm 8 Method to check if user has permission

1: **procedure** $hasPermissions$($right$,s,$policies$)
2: $newList \leftarrow [read, write, writeplus, own]$
3: $count \leftarrow 0$
4: **for** each entry i in $policies$ **do**
5: **if** $i.subject == s$ **then**
6: $newList \leftarrow intersection(\text{i}.rights, newList)$
7: $count++$
8: **end if**
9: **end for**
10: **if** $count == 0$ **then**
11: $return false$
12: **else**
13: **if** $newList.isElement(right)$ **then**
14: $return true$
15: **else**
16: $return false$
17: **end if**
18: **end if**
19: **end procedure**

2016 **Access Control in Weakly Consistent Systems** Tiago Costa