Mutida: A Rights Management Protocol for Distributed Storage Systems Without Fully Trusted Nodes

Bastien Confais, Gustavo Rostirolla, Benoît Parrein, Jérôme Lacan, François Marques

Mutida: A Rights Management Protocol for Distributed Storage Systems Without Fully Trusted Nodes

Bastien Confais[1], Gustavo Rostirolla[1], Benoît Parrein[2], Jérôme Lacan[3], and François Marques[1]

[1] Inatysco, 30 rue de l'Aiguillerie, 34000 Montpellier, France
{bastien.confais,gustavo.rostirolla,francois.marques}@inatysco.fr
[2] Nantes Université, Polytech Nantes, rue Christian Pauc, BP50609, 44306 Nantes, France
benoit.parrein@univ-nantes.fr
[3] ISAE Supaero, 10, Avenue Édouard-Belin, BP 54032, 31055 Toulouse, France
jerome.lacan@isae-supaero.fr

Abstract. Several distributed storage solutions that do not rely on a central server have been proposed over the last few years. Most of them are deployed on public networks on the internet. However, these solutions often do not provide a mechanism for access rights to enable the users to control who can access a specific file or piece of data. In this article, we propose Mutida (from the Latin word "Aditum" meaning "access"), a protocol that allows the owner of a file to delegate access rights to another user. This access right can then be delegated to a computing node to process the piece of data. The mechanism relies on the encryption of the data, public key/value pair storage to register the access control list and on a function executed locally by the nodes to compute the decryption key. After presenting the mechanism, its advantages and limitations, we show that the proposed mechanism has similar functionalities to Wave, an authorization framework with transitive delegation. However, Wave does not require fully trusted nodes. We implement our approach in a Java software program and evaluate it on the Grid'5000 testbed. We compare our approach to an approach based on a protocol relying on Shamir key reconstruction, which provides similar features.

1 Introduction

Currently, there are several distributed storage solutions that either rely on a central metadata server used to locate the data replicas or use a peer-to-peer (P2P) protocol that is suitable for deployment on public networks with nodes that are not necessarily trusted. When a trusted metadata server is present, it is relatively easy to manage the access rights [41]; the server checks if the user is allowed to access the file before distributing the location. In a full P2P network [38, 25], with untrusted nodes and pieces of data that are publicly accessible, managing access rights poses a different challenge. The main challenge

is that there are no servers that we can rely on to be in charge of the protocol for rights management. Additionally, anonymity is a key point since peers are communicating directly and additional layers need to be in place to guarantee it. [19]. In this paper, we propose Mutida (from the Latin word "Aditum" meaning "access" and written from right to left), a protocol that focuses on enabling users to manage access rights in a network of the second category.

In this paper, we consider the files stored on the Interplanetary File System [11] (IPFS), which is a storage solution that relies on a BitTorrent-like [35] protocol to exchange the files between the nodes and on a Kademlia [39] Distributed Hash Table (DHT) to locate the different pieces of data in the network. The essential characteristic of IPFS is that files are immutable and cannot be modified once they are written.

Using a DHT forwards all location requests through different nodes. Therefore, any connected node is involved in the routing of requests and can determine the identifiers of popular files [26]. Additionally, since the storage solution does not manage any permission and because IPFS does not provide any encryption mechanism to protect user privacy [47], every user can access all the files stored in it if the content identifier (CID) is known.

The consequence of this is that any node in the IPFS network can observe the DHT requests and access the corresponding files. This illustrates and justifies the need to manage access permissions in such a network. With Mutida, all users will still be able to find the files in the network, but only the permitted users will be able to decrypt the content.

Because of the use of Merkle trees [40] and the use of a root hash as a file identifier, users of IPFS do not need to trust the storage nodes to be assured that the retrieved file has not been tampered with. In our protocol, to manage access rights, we rely on the same level of trust; the exchanged messages are protected against any disclosure to a third party and are not corrupted on the path. However, the nodes themselves can be malicious and cannot necessarily behave as expected. They may go offline with no warning. One essential requirement of our proposition is that the owner of a file can delegate its access permission to another user. Similarly, computing nodes should be able to access and decrypt pieces of data when a user requests them to process their own data.

In Mutida, the files are encrypted before being stored in the distributed data storage solution. Then, a public Access Control List (ACL) and a local function that can be executed by each client are used to determine the key required to decrypt the file. The ACL consists of key/value storage deployed in the same network. The values are public, and the modifications can be controlled by trusted nodes or consensus algorithms. We also mention that a detailed security analysis of the proposed mechanism is beyond the scope of this paper. Additionally, data access revocation is contemplated in Mutida but not guaranteed. Ensuring revocation in a distributed manner is explored in [32, 14] or legally in [33]. The main contributions of this article are as follows:

- a method for data access control based on a function executed locally on the client;

– a delegation mechanism to distribute access rights to users or to the nodes that we want to allow permission to process the data;
– a performance comparison with a solution based on key splitting, including the impact of network limitations on each method.

It is the protocol to manage decryption keys and delegations, thought it does not directly provide functionalities of authentication, authorization verification or accountability. Our approach uses common cryptographic functions to build the desired features, and its novelty resides in the way that these functions are combined to form a new protocol for right management.

The remainder of this paper is organized as follows: In Section 2 we present the related work. In Sections 3 and 4, we introduce the usage scenarios and the Mutida model, followed by Sections 5 and 6, where we provide the methodology and the results obtained. Finally, in Section 7, we present the paper conclusions as well as directions for future works.

2 Related Work

The majority of the approaches that deal with the problem of managing access rights in a distributed environment rely on data encryption. This problem is relevant in a wide variety of domains, such as healthcare [31], data sharing [3] and administrative environments [17]. Several approaches, including a subset that is detailed below, can be found in the literature. However, to the best of our knowledge, none of the proposed methods allows an access right delegation mechanism in a P2P manner with a specificity to grant compute nodes a temporary permission to access data on behalf of the user requesting the computation. In our case, we follow the delegation definition of Gasser and McDermott [21], which describes the process where a user in a distributed environment authorizes a system to access remote resources on their behalf. We also highlight that in most of the works where the file key is exchanged through re-encryption using a public key, such as [29, 30] and [51], the key is generally known by a given group, and thus, the file owner could give access even without the users consent or demand.

In 2008, Jawad et al. [29] proposed a solution where files are stored in encrypted form. The user must then communicate directly with the owner of the data to obtain the key. The clear limitation of this approach is that it requires the presence and simultaneous connection to the network of the owner of the piece of data and the user wishing to access it. This constraint is present in many other propositions, as emphasized by Yang et al. [61]. Moreover, sometimes the key exchange involves a trusted third party [3]. Adya et al. [1] remedied this constraint by proposing to create a data replica per user. Giving authorization to a user to access the data entails creating a new replica encrypted with the user's public key. As a result, the user and the owner do not have to meet to exchange keys, but the price to pay is a substantial increase in the use of storage space.

Another proposal is to manage the keys within a blockchain [54, 10, 52] instead of a trusted third party. For Steichen et al. [54], the blockchain was used to store the access control list. Storage nodes were responsible for consulting it before distributing data to the user. The major disadvantage of the approach was that it assumed that the storage nodes are trusted enough to not deliver data to unauthorized people. Similarly, Battah et al. [10] proposed the addition of a multiparty authorization (MPA) scheme, which was also stored in a smart contract to ensure that a single malicious party could not act alone. The consequence of this scheme was that the whole approach relied on proxy re-encryption nodes associated with a reputation scheme, as well as shared keys among the parties where a minimum number must be collected to access the file decryption key, thus increasing the complexity and time for the exchanges to take place.

To overcome this, Sari and Sipos [51] proposed an approach where data is encrypted with a symmetric key. This symmetric key is encrypted with the user's public key and stored in the blockchain. Xu et al. [60] corrected the trust problem by not only using the blockchain to store the access control list but also by implementing the verification of access rights within smart contracts. The idea is that the nodes of the blockchain come to a consensus on whether a user can access the requested data and issue them the key. The disadvantages of such an approach include the induced latency due to the use of a distributed consensus and the blockchain being an append-only data structure. Thus, it can be a space problem when access changes regularly: new users are allowed, and others have their permission revoked. Attribute encryption [58] and proxy re-encryption [15] are also two other approaches that have been proposed. The first required complex key management and the second required trust in the machine that adapted the encrypted data to the user's key.

Alternatively, broadcast encryption [30] is an encryption technique that consists of encrypting content for a group of users. Each user has a unique set of keys. A set of keys is used in encryption that allows only a specific group of users to decrypt the data. This system works well when there are few different groups and each group has numerous users. In this case, the number of managed keys is lower than the classic solution using ACLs. In addition, each piece of data is accessible by a unique group of users, which would lead to the use of a large number of keys. The main flaw shared by most of these solutions is that they do not allow permission delegation. A user who has obtained the rights to the owner's data cannot authorize a machine to access this data as part of the execution of a computation. The other limitation is anonymity; access control lists make it clear who can access what data.

Beyond encryption to manage access rights, some protocols are dedicated to key management and delegation. Lesueur et al. [36] proposed building a Public Key Infrastructure (PKI) in a P2P manner. Their protocol relies on key splitting and partial signatures. This was a major advance in the sense that it enabled distributed decisions to be made. For instance, nodes can agree to sign the public key of a new user so that it can be trusted or to sign any request that requires a consensus, such as a request to access a certain piece of data. The

major drawback of this proposal is that it is difficult to manage redundancy in key parts and to react when a certain number of nodes leave the network simultaneously. This idea of distributed signatures was used by Wang et al. [59] to manage the access rights of data.

Some articles proposed protocols considering specific problems of the right management in a distributed solution. For instance, Tran, Hitchens and Varadharajan [28] considered trusting the nodes because some nodes can act in a malicious way. Similar to our context with data immutability, we have the content protection of recordable media (CPRM) [23], where the data cannot be modified. However, the main difference between the protocols is that in our approach, the access rights should not be given without the request of a user.

Wave [4] is one of the rare protocols that focuses on permission delegation in distributed applications. The protocol has the specificity that it does not require any centralization by relying on a blockchain-like solution to store the access rights. Nevertheless, Wave is more focused on rights management for applications. In their case, the nodes that provide the service must check if the client has the right to access a given service. While that is feasible in the service context, it would be a blocking point for storage components such as IPFS, meaning that it would need to be modified to verify the rights before delivering the content. For us, the nodes do not deliver a service besides data, and therefore, we cannot trust these nodes to manage the rights.

As in our protocol, Wave allows users to create delegation chains with an anonymity on the created delegations. When a user requests a permission to a node, the node granting the permission sends a record to the user and stores a second record encrypted in the blockchain. The external users cannot determine the permissions by reading the blockchain content, but a specific node to which the users send the record is able to verify the validity of the chain delegation from the blockchain. The difference from our protocol and its main drawback is that the permission check is performed by the node delivering the service. This difference implies that Wave requires that the nodes delivering the services be trusted nodes. Some proposals, such as Aura et al [7], used a more straightforward implementation using the signature of certificates, similar to what it is used in Public Key Infrastructure.

Access revocation is also an entire topic to discuss. Revocation has always been a difficult problem in distributed solutions. One of the best examples of this is that certificate revocation in browsers trusting different authorities still does not have an ideal solution [16]. In distributed storage solutions, common solutions rely on a distributed consensus [34] and generally use a blockchain, but it is sometimes not enough for a single node to make the decision to deliver the piece of data to a user. In this case, Schnitzler et al. [52] proposed an incentive to the nodes to revoke the access and delete the pieces of data when needed, but it does not guarantee that Byzantine faults are avoided.

In other commercially available approaches for authentication and authorization, such as Kerberos [41] and Oauth [24], the user contacts a server that delivers a token used to access different services. The server delivering the token

can be seen as an ACL server (similar to the one described in Section 4), and then the token is used to connect different storage nodes that can send the data. However, these models imply trust in the server delivering the service.

Finally, some papers focused on the problematic of anonymity. This means that the nodes should not be able to establish a list of the files a user can access. Backes *et al.* [8] proposed such a solution where nodes can only determine if the user sending a request is allowed to access the piece of data or not, without revealing any piece of information about the user.

We propose Mutida to fill the gap with a method that allows an access right delegation mechanism in a P2P manner with a specificity for delegations to compute nodes that have a temporary permission to access data on behalf of the user requesting the computation. Our method allows the management, delegation and revocation of rights over a file in a distributed P2P system. The goal is to allow the users to recreate all the file keys that they have access to with a single local function and a key pair. The usage scenarios and assumptions for Mutida, as well as a detailed model description, are presented in the following sections.

3 Usage scenarios and assumptions

The first use case we target is a user who stores their own data on their own IPFS node. Assuming that the user wants to be the only person to access their data, the right management protocol should protect the data against unauthorized access and should not have a strong overhead.

The second use case targeted is when the user wants to be able to share the data with another user. The other user sends a request to the owner to obtain permission to access the file. The owner accepts the request and enables this second user to compute the decryption key of the file.

The third use case is a situation where a user that is allowed to access a piece of data should be able to ask a computing node for processing the data. For this, the user should be able to temporarily give permission to the compute node to access and decrypt the file. These two last use cases indicate that the solution should have a delegation mechanism with the following properties:

i) The owner of a piece of data is the only user who decides which user can access it.
ii) A delegation can be made only if the user makes a request to the owner to access it.
iii) Users would be allowed to access a file even if the owner is not currently connected to the network.
iv) A delegation can be temporarily established from a user to a computing node.

4 Mutida Model

The Mutida method enables the management, delegation and revocation of file permissions. The goal is to allow the users to recreate all the file keys they have access to using a single local function and a key pair (each user possesses a private key pair $Kpriv_{User}$ and $Kpub_{User}$), with the assumption that the people we want to exchange with know the $Kpub_{User}$ value. Additionally, access to a piece of data must be possible even if the owner of this piece of data is not online. The solution also allows rights delegation to a third-party node for the execution of a specific task, such as data processing.

Our solution relies on a global ACL, similar to that of Wang et al. [58], which consists of key/value storage deployed in the same network. The values are public, and modifications are controlled by trusted nodes or consensus algorithms. This enables us to manage the file permissions without a centralized authority and even when the file owner is offline. We also rely on two local functions called $ID1$ and $ID2$, which are known by all the peers in the networks and will be detailed later in this section.

We summarize the assumptions of the Mutida protocol as follows:

i The files are stored in a public server and are publicly available.
ii Each file has a unique identifier: a unique filename or a UUID.
iii Each user has a secret key used as an input of the Mutida "ID" functions.
iv The ACL is centralized or distributed key/value storage. Records can be read by everybody, but modifications are not possible or are controlled by the key/value storage.
v The network exchanges are encrypted so that an adversary cannot intercept the messages and gain unauthorized access to files.

Figure 1 illustrates the network on which Mutida is deployed, with the IPFS storage system deployed at multiple locations and ACL nodes spread on each site. We assume that users connect to the closest node to store their files. As the main advantages of the Mutida approach in comparison with a standard approach, which consists of encrypting the file key with the public key of each user that wants access to the file, we can list the following:

i) Deleting a record in the ACL removes the access right for a user and for the computation nodes to which this right has been delegated. In a version using more traditional cryptography, this dependency is nonexistent.
ii) It is not possible to give access permission that a user has not requested. In Mutida, each delegation begins with the exchange of IDs that only the user receiving the delegation is able to calculate. In contrast, in a more "traditional" approach, knowing a user's public key is enough to create a record in the ACL, and therefore, grant rights. Although this can be overcome by using digital signatures, our approach integrates this functionality natively.
iii) For the performance of the RSA calculation or what Shamir compared to the additions of Mutida, in our case, the calculation is limited to a simple addition, whereas an asymmetric RSA-type encryption requires exponentiation. Quantitative data to justify this point are presented in Subsection 6.1.

iv) As a result of using a hash function, we have anonymity in the ACL; it is not possible from a record to determine which user and which file it corresponds to. In an approach using "classic" cryptography, there is no consensus to achieve such functionality.

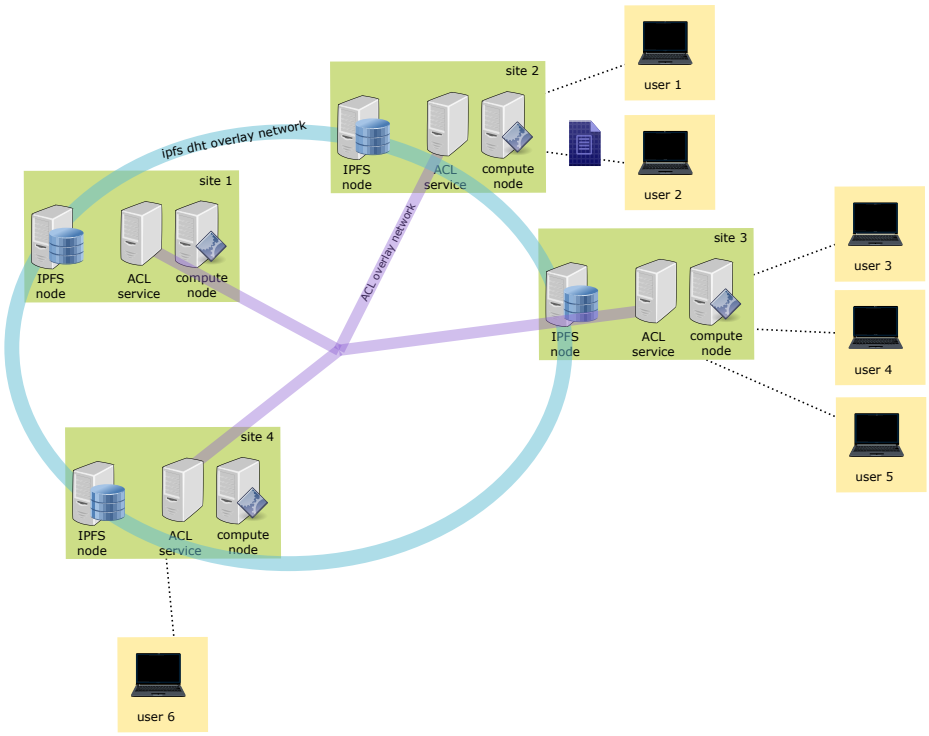


Fig. 1: Overview of the Mutida architecture.

4.1 Protocol description

Hereafter, we describe the main operations, as shown in Figure 2, that allow us to manage the file rights without a centralized authority and leverage from a distributed storage system. As the main requirements of the proposed protocol, we highlight the following:

i) The owner of a file chooses who can access it and delegates the right to selected users.
ii) The users that can access a piece of data can temporarily delegate their access rights to a computing node to execute some calculation.

iii) The owner should not be able to give to a user the permission to access a file if the user has not requested it.
iv) The allowed user should be able to access a file even if the owner is not currently connected to the network.
v) The revocation of access rights should also be available, even if not guaranteed, despite the nature of the storage solution.

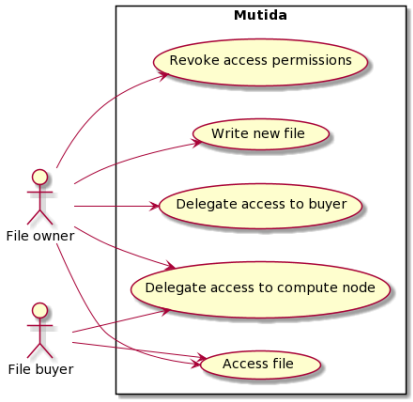


Fig. 2: Diagram of possible user actions in Mutida.

Writing a new file. Figure 3 shows the sequence diagram when a user wants to store a new file. The operation is divided into two phases. During the first phase, the user determines an encryption key and encrypts the file. The user selects a random value R_{owner}, denoted R in the diagram, and computes the key with the help of a previously agreed ID1 function, as the one described in Equation 1. In the remainder of this text and equations, we refer to file identification as "filename" for simplicity, but for implementation purposes, a Universal Unique Identifier (UUID) of each file should be used.

Because the SHA256 function returns 256 bits, all the computations described below use binary words of that size. In other words, all the computations are modulo 2^{256}. This function is chosen due to its low collision probability and computing complexity, as we lack a correlation between the input and the output bits [18, 45].

From the value computed in Equation 1, the final value used as a key for the file would be according to Equation 2.

$$\begin{aligned} &ID1(user\_private\_key, filename) \\ &\quad = SHA256(concatenate(user\_private\_key, filename, \text{“}ID1''\text{)}) \end{aligned} \tag{1}$$

$$file\_decryption\_key = (ID1(owner\_private\_key, filename) + R_{owner}) \ mod \ 2^{256} \tag{2}$$

We note that in Equation 1, the value ID1 between quotes corresponds to the actual word (it is not a recursive function). This string is used to create different ID functions (ID1 and ID2) where the values are not correlated between them. These functions are easy to compute [48] by the user who knows the private key, though they appear completely random to others.

After encrypting the file, the user should store the value R_{owner} in the global and the public ACL (key/value storage). This is the second phase of the operation. Because the ACL is public key/value storage, the ACL key should be carefully chosen. The obvious solution is to use a couple ("user1", "file1"), but this couple has the major drawback of making the system transparent; every user can determine the files that can be accessed by anybody. To overcome this, we propose to compute the ACL key using Equation 3. Therefore, an observer could not determine the user or the file that the record is for.

$$\begin{aligned} &ID2(user\_private\_key, filename) \\ &\quad = SHA256(concatenate(user\_private\_key, filename, \text{“}ID2''\text{)}) \end{aligned} \tag{3}$$

A signature is also added to the ACL record. It will enable the user to determine if the stored value has been modified or corrupted when it will be retrieved and allow verification of the user's the identity that created this record. The signature is not stored directly in the ACL because it can break the anonymity or the privacy of the user. Instead, we perform an XOR operation (noted $\oplus$) between the signature and the hash of the decryption key to ensure that only users who know the decryption key are able to extract the signature.

By computing the encryption key in the aforementioned way, we ensure that:

- If someone reads the public ACL and accesses the value R_{owner}, they cannot determine the decryption key because they cannot compute the value of $ID1(owner\_private\_key, filename)$ without the private key of the user.
- if someone reads the public ACL and knows the decryption key of a file (because it is allowed to), they cannot determine the private key of the user.
- If someone knows the value of $ID2(owner\_private\_key, filename)$, they cannot determine the key for the other files that the user can access.
- It is not necessary to keep a local keystore of all the files that the user has access to.

In Figure 3, for illustrative purposes, we describe Equation 1 as the function $ID1(owner\_private\_key, filename)$. In the figure, $user1(owner)$ creates a new ACL entry for $file1$ with the value 20, and the file encryption key would be the value 20 + 4497, where 4497 corresponds to their own $ID1$ value calculated using Equation 1. Similarly, the computed value for the ACL using Equation 3 is 2021.

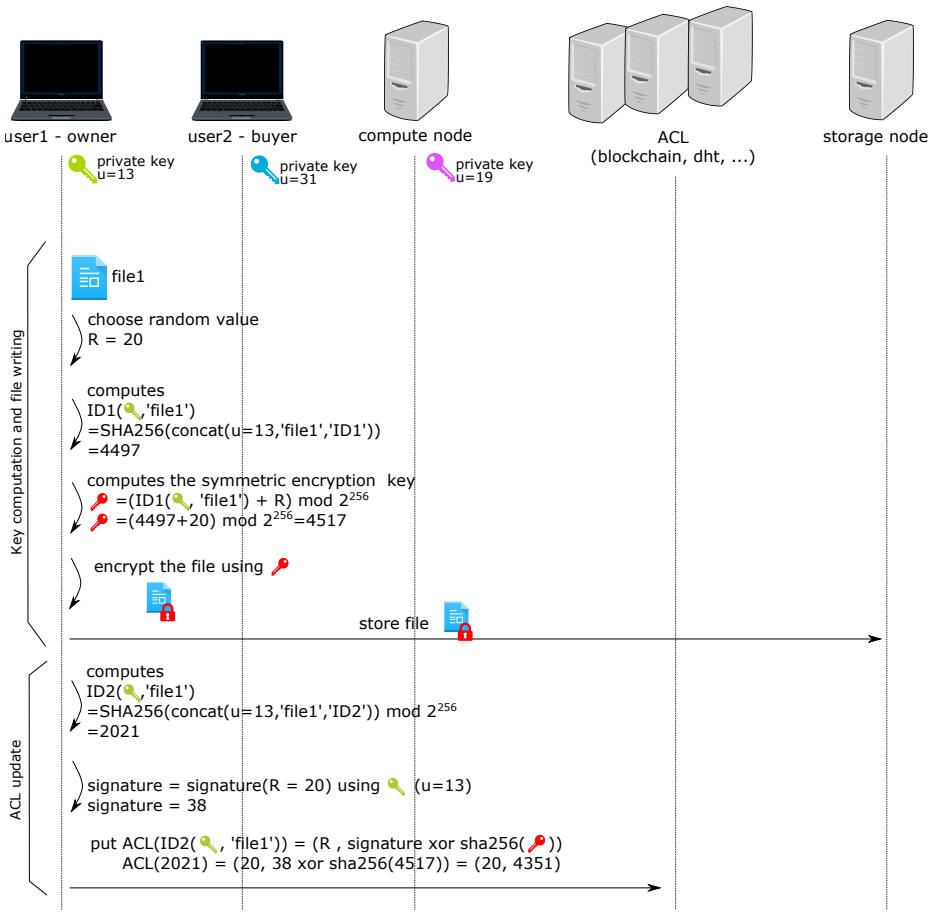


Fig. 3: Sequence diagram describing the creation of a new file.

Accessing a file as an owner. The access of a file as an owner is illustrated in Figure 4. The operation is divided into 4 phases: retrieving the value stored in the ACL, computing the decryption key, checking the integrity of the value retrieved in the ACL, and finally, accessing the file and decrypting it.

To access the file, the user must first retrieve the value stored in the global ACL. The user first computes the value of the ACL key using $ID2(owner\_private\_key, filename) = 2021$. Afterward, the node retrieves the couple of values (Equation 4) associated with the key.

$$R_{owner}, signature \oplus \text{SHA256}(key) \tag{4}$$

With the value R_{owner}, the user computes the decryption key with the same formula as previously presented in Equation 2. This allows us to recalculate all the keys for the files that we own or have access to without having to store any additional value locally. Once the user has recalculated the key, they only have to retrieve the file from the IPFS public storage and decrypt it.

With the decryption key, the user is able to extract the signature from the value retrieved from the ACL and check if the R_{owner} value is not altered in the ACL storage system. In other words, the user can check that the computed decryption key is correct and can retrieve the file and decrypt it.

In Figure 4, $user1$ retrieves the previously stored value in ACL and is able to reconstruct the key just using this value and the value obtained by their $ID1$ function. The same would apply for multiple files, without the need to have a local keystore for each file that belongs to the user.

Access delegation to another user. The idea of the right delegation is to enable another user (called "buyer") to decrypt the file without re-encrypting it (we restate that IPFS stores immutable pieces of data). Therefore, the user who gains access to it will have to be able to compute the same decryption key as the owner, but using their own private key.

To accomplish that, the user has to request access to the data, as shown in the first phase of Figure 5. The user uses Equation 1 to compute a value that is then sent to the owner of the file. Because the value is sensitive, as it enables any malicious user who could learn it during the exchange to later be able to compute the decryption key, the user adds a random number (noted k) to it before sending it. In other words, the buyer sends the value $(ID1(buyer\_private\_key, filename) + k)\ mod\ 2^{256}$ to the owner of the file. The value k also enables the protocol to work in an asynchronous way by posting the request in a public queue that is processed once the owner of the file is online.

If the owner of the file agrees to give access, they retrieve the ACL value (second phase) and compute the delta+k value, which is the difference between the decryption key and the value sent by the buyer (3rd and 5th phases). The $delta$ value is computed as in Equation 5.

$$file\_decryption\_key = (ID1(buyer\_private\_key, filename) + k + delta)\ mod\ 2^{256} \tag{5}$$

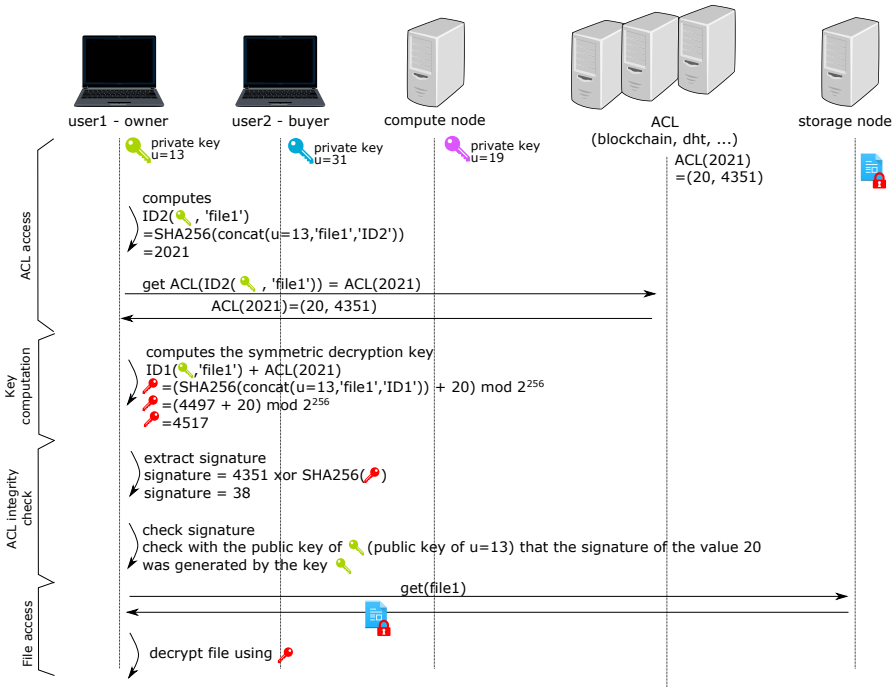


Fig. 4: Sequence diagram describing the access of the file by the owner.

In this computation, the owner cannot determine the private key of the user they give the permission to because of the use of a hashing function and the random value added to it. The value looks random to the owner, but it can be computed easily by the buyer. In the same way, the owner cannot use the value transmitted by the user to access the other files that this user is allowed to access because the $ID1$ value depends on the "filename" and the k value is unknown to the buyer. In the last two phases, the value *delta* is returned to the user who removes the random value k, as in Equations 6 and 7

$$R_{buyer} = (delta + k) \ mod \ 2^{256} \tag{6}$$

$$R_{buyer} = (file\_decryption\_key - ID1(buyer\_private\_key, filename) - k + k) \ mod \ 2^{256} \tag{7}$$

Finally, the user stores the value R_{buyer} in the ACL using the same mechanism as previously described, computes the key of the ACL record using the function $ID2$ and adds a signature to protect the record against any modification.

In Figure 5, we illustrate the file access delegation process where $user2(buyer)$ sends a request to access $file1$ that belongs to $user1(owner)$, accompanied by their ID value (3951). To delegate the rights, $user1$ calculates a delta between the file decryption key and the ID1+k value of user 2 (561). $User2$ removes the k value and stores the R_{buyer} value (566) in a new ACL entry.

Accessing a file as a delegated user. The user accesses the file using the same process previously described for the owner, where the single change is the value that is read corresponding to this user. The user has to read the value in the ACL, as in Equation 8, and then compute the key with the same formula as previously presented, where
$key = ID1(buyer\_private\_key, filename) + R_{buyer}$. Finally, the user can extract the signature, verify it and retrieve the file from the IPFS public storage before decrypting it

$$\begin{aligned} &ACL(ID2(buyer\_private\_key, filename)) \\ &= (R_{buyer}, signature \oplus SHA256(decryption\_key)) \end{aligned} \tag{8}$$

In Figure 6 we illustrate the file access by $user2$. The process starts by recovering the corresponding value in the ACL (566) and adding it to their own ID1 value (3951), which will result in the decryption key of $file1$ (4517).

Access delegation to a computing node. A specificity of our approach is that a user can request a computing node to execute a software program that uses the user's data. The user can enable the computing node to access the pieces of data on their behalf to perform the requested computation.

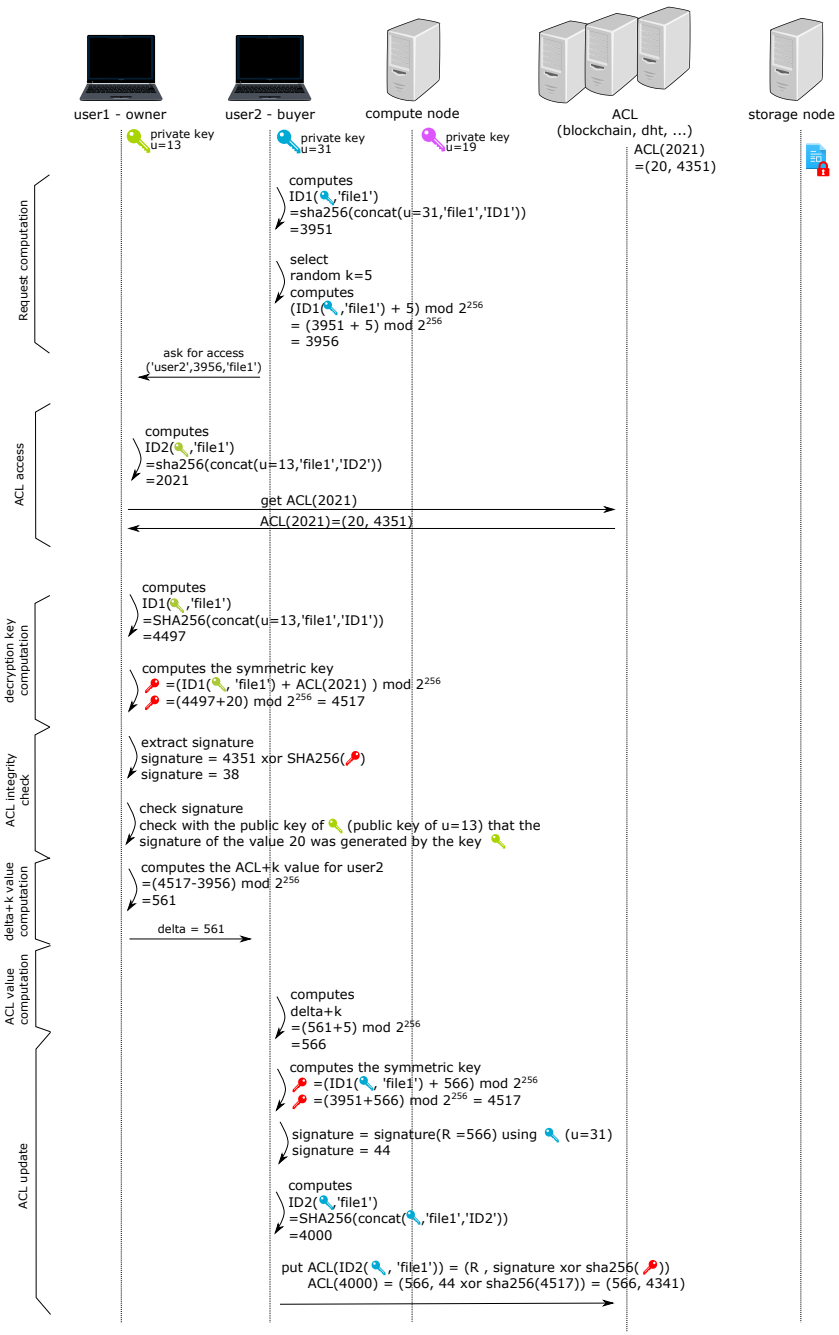


Fig. 5: Sequence diagram describing the rights delegation to a new user.

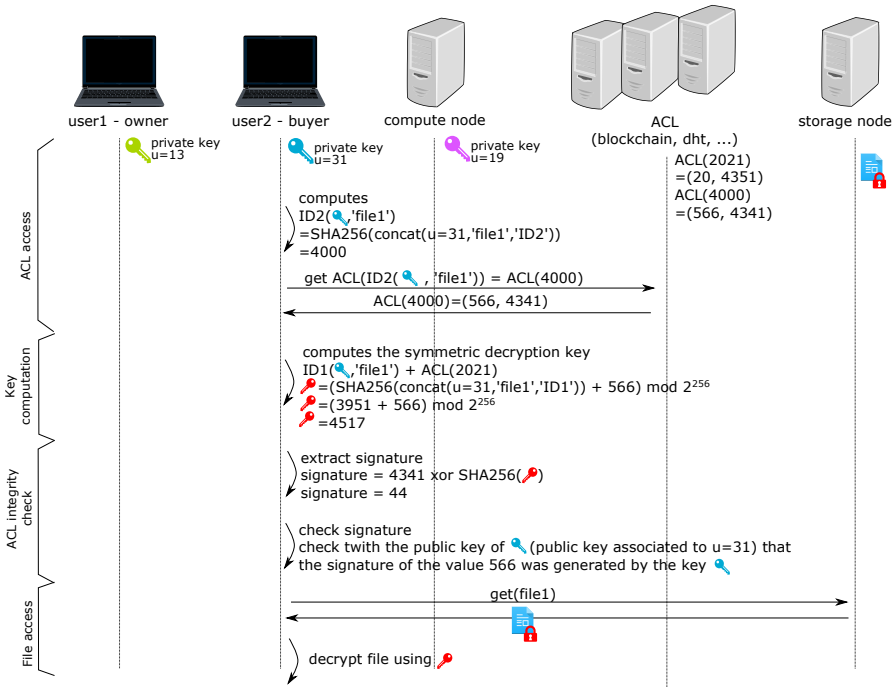


Fig. 6: Sequence diagram describing the access of the file by a delegated user.

The ideal approach would be to use fully homomorphic encryption [13] on the compute node so that the computation will be performed directly on the Encrypted pieces of data. However, due to the lack of maturity and the need for an operational solution, our delegation mechanism enables the node to compute the key and decrypt the pieces of data.

Delegating permission to a computing node is quite similar to delegating permission to a user. The difference is that instead of storing the value in the ACL, the user transmits it to the node directly. The idea is that the computing node does not need to access the files over a long period of time. The computing node can forget the value once the process requested by the user is terminated. Additionally, if at any moment there are changes in the permission for a given file, the same will be replicated for the computing node.

Our protocol does not guarantee that the computing node will delete the key after the computation is finished. However, to the best of our knowledge, the only way to ensure that there is no replica of the key is to utilize an encryption scheme that, according to Naehrig et al. (2011)) [44], is "somewhat" homomorphic, where we would support a limited number of homomorphic operations that can be much faster and more compact than fully homomorphic encryption schemes. While this could solve the issue of the user knowing the key, only a fully homomorphic scheme can prevent copies of the unencrypted file when we want to perform any kind of computation on it. Until this moment, this kind of encryption scheme is unfeasible due to its poor performance, as indicated by Fontaine and Galand (2011)) [20]. An alternative approach and the only one that seems feasible at the moment would be to rely on the legal side of the General Data Protection Regulation (GDPR) as proposed by Kieselmann et al. (2016) [33] and Politou et al. (2018) [46].

To start the process, the user requests the computing node to send the value of $ID1(compute\_node\_private\_key, filename) + k$, where k is a random number. As in the delegation between two users, the k value prevents any leak of the $ID1$ value and enables the protocol to work asynchronously.

Then, the user computes S according to Equation 9, where R_{buyer} is the ACL value for the user. Finally, the user transmits the delta value to the compute node, as well as the ACL key by performing $ID2(user\_private\_key, filename)$ to the compute node.

$$S = decryption\_key - (ID1(compute\_node\_private\_key, filename) + k + R_{buyer}) \tag{9}$$

In Figure 7, we illustrate the rights delegation process from $user2$ to a computing node, where the computing node should use the same entry as the corresponding user. We start by asking for the ID of the respective computing node (5643) that is added to a random number k to protect the node against any leak of the ID1 value. Then, R_{buyer} is computed, which is obtained by the difference between the ID1 of $user2$ and the computing node's $ID1 + k$, obtaining (-1554).

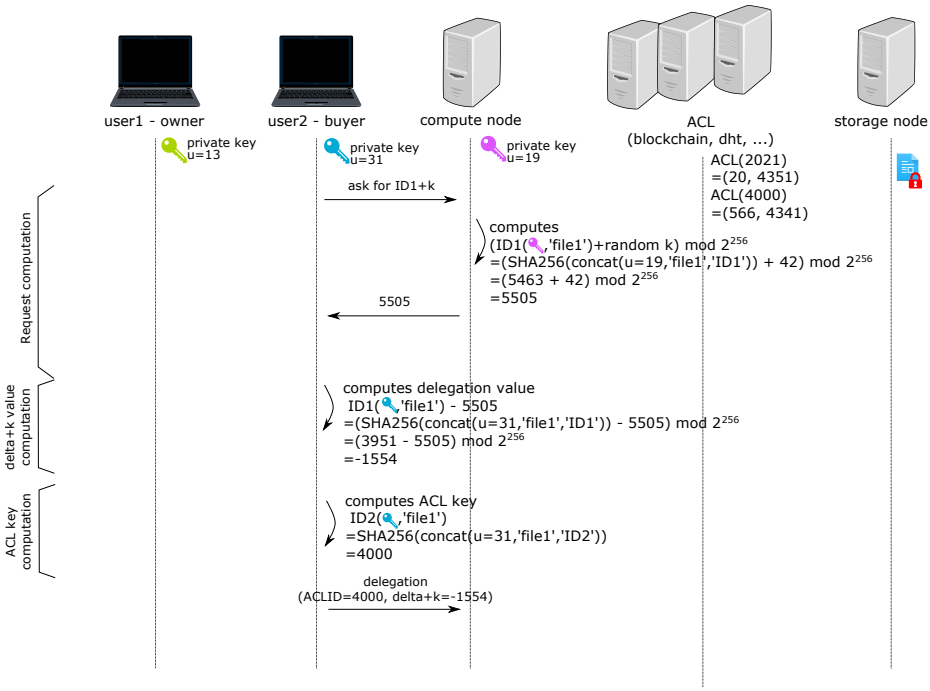


Fig. 7: Sequence diagram describing the delegation to a computing node.

Accessing a file from a computing node on behalf of a user. To access a file, the computing node will read the ACL value of the user: R_{buyer}, then it computes Equation 10, where S is the value transmitted by the user at the end of the delegation process. Because the node accesses the ACL value of user R_{buyer}, if the user has their access revoked and the ACL value is deleted, the computing node cannot compute the key and decrypt the files. We also note that the delegation is on a file basis. Therefore, the computing node cannot access all the files the user has access to.

$$file\_decryption\_key = ID1(compute\_node\_private\_key, filename) + R_{buyer} + S + k \quad (10)$$

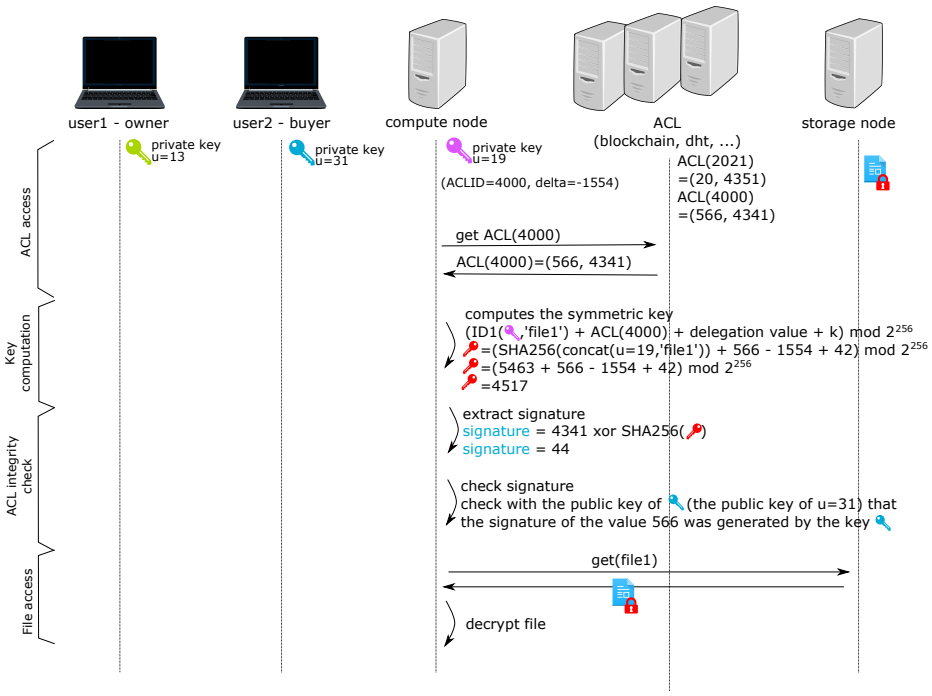


Fig. 8: Sequence diagram describing the access of the file by a computing node.

In Figure 8, we show the computing node utilizing the previously calculated R_{buyer},as well as its own $ID1$ value and the one in the ACL to recalculate the file key. As in the previous scenarios, the node can verify that the value retrieved from the ACL has not been corrupted.

4.2 Storage of the global ACL

The main characteristic of the ACL is being global and public. This means that each node should be able to access the values stored in it. The idea of using a public Access Control List is not new and it is used in different articles [37, 2]. Several implementations can be evaluated to store this Access Control List, such as Distributed Hash Table (DHT), Blockchain, DNS and Gossip-based systems. Each of these solutions has some advantages and drawbacks. We attempt to evaluate them in the following sections.

Distributed Hash Table. A distributed hash table is a key/value data storage spread among several nodes. The nodes are organized around a virtual ring, and routing tables guarantee that each value can be retrieved by contacting at most $log(N)$ nodes (with N, the number of nodes). Different variants of distributed hash tables exist, such as Chord [55], Tapestry [63] and Kademlia [39], which attempt to optimize the routing process.

To store our ACL values, a DHT has many advantages. First, it guarantees that each value can be found by contacting a limited number of nodes, which leads to good performance. It also has the advantage of evenly spreading the values among all the nodes of the network. There is no node that stores more keys, and therefore, has more power to control the network.

However, there are two main drawbacks. The first is a classic drawback of a DHT; replication is not part of the protocol and should be managed on top of the DHT. Therefore, there is a risk that some keys are lost when nodes disconnect from the network. The other drawback is more important; a node that is responsible for storing a key can do everything with the key: it can delete it, modify the value or refuse to serve it to some nodes.

In other words, DHT does not natively support Byzantine faults and malicious nodes [57]. This is important for our protocol because if the owner of a certain file allows a second user to access it, a new key is inserted in the ACL. However, if the node storing this value does not let the user access it, it means that the ACL node has the power to limit the access beyond the will of the data owner.

Domain Name System. A domain name system [42] (DNS) is a distributed database specifically used on the internet to associate IP addresses with domain names. The particularity of the protocol is to use a hierarchical namespace, such as "key.domain.".

This hierarchical organization leads to the spread of the workload and storage among different servers. This protocol has the same drawbacks as the distributed hash table (DHT), but it proposes a deterministic network routing. It also lacks automatic reconfiguration in the case of network modification [27]. For instance, when a node is added, an administrator has to create the DNS records to attach the new node in the tree. However, in the situation of ACL distribution, this protocol can be a solution in the case of some trusted nodes managing the top of the hierarchy, preventing users from being unable to access the records.

Blockchain. Another possible implementation for the key/value storing the ACL values is to use a public [61] or a private [5] blockchain. Blockchains are immutable data structures that work only as "append-only", which is replicated on all the nodes. Therefore, compared to a DHT, there is no risk that a user cannot retrieve a piece of data stored in it. In the situation of distributing the Access Control List, this property is important because it means that all of the nodes store a copy of it and no node is able to prevent any user from accessing it.

In addition to the data structure, blockchains provide a consensus algorithm. Each transaction is validated by a majority of nodes before being added to the blockchain. Therefore, any action of adding or modifying a value is not taken by one node in particular. The main drawback of a blockchain is the computing power to achieve a consensus. To overcome this, some proposals replace the consensus based on proof-of-work [61] with other types of proof, such as proof-of-stake [50] and proof-of-authority [6].

Another way to overcome this is to use a private blockchain. A private blockchain is one where nodes need the permission to participate. The nodes must be trusted and should not be malicious. The other disadvantage of blockchains is that no value can be deleted because of the append-only structure of the chain. Therefore, in our situation, it makes the revocation of access rights impossible.

Discussion. From the previous discussion, the choice to store the global ACL can be seen in the following order of preference:

i Blockchain is first because of its ability to make the ACL available across all nodes. It is also possible to deploy a hyperledger on all nodes that would manage ACLs in public transactions and handle the users who are allowed to join the network in a distributed way. Traceability can be managed by private transactions stored on trusted nodes.
ii DHT is next because of its ability to dynamically adapt to the network. Furthermore, this technology is already used in IPFS. We can imagine deploying this solution by inserting keys manually into the DHT of IPFS.
iii DNS is third because of its tree structure and performance. The root nodes of the tree can be managed by the trusted certifier nodes.

4.3 ACL management

These different systems do not always provide strong consistency. Therefore, two simultaneous reads on the same record can lead to reading different values if the record was recently updated. This is particularly true in blockchains when new blocks have not been propagated to all the nodes or to the DNS when the zone was not updated on the secondary servers.

We believe this is not a real problem because there is a record for each user. Therefore, there are no concurrent reads between users. The second reason is that because data are immutable, ACL records do not vary much. For a user and a specific file, there are two possibilities: either the record is here and the

user will be able to decrypt the file or the record is not here, which means that the user's right has been revoked.

In the worst scenario, the user that just received the permission cannot still decrypt the file or the user that just saw their permission revoked, though they can still access the file. There is no situation where the user computes the wrong key.

The second point is the security of the ACL. We previously described how a signature can help to determine if the record was tampered with, but it does not prevent tampering itself. There are two ways of managing this. If right revocation is not wanted, the ACL storage system can be a system in append-only mode. Therefore, no modification of the ACL is needed.

Otherwise, there must be a trust between the user and the ACL storage system that will need to verify some permissions. A simple way to manage the permission is to use a token that will be specified at the creation of a record and that must be given to delete it.

5 Methodology

An implementation of all the necessary Mutida components described in the previous sections is performed in Java Spring Boot. We rely on the standard MessageDigest library and $SHA-256$ for the $ID(user, filename)$ function implementation. Each client is composed of a REST API with all the encryption, file and ACL endpoints, as well as an IPFS [49] peer for data storage. The platform is deployed using Kubernetes [56], where all the nodes allocated form a single cluster. This is illustrated in Figure 9.

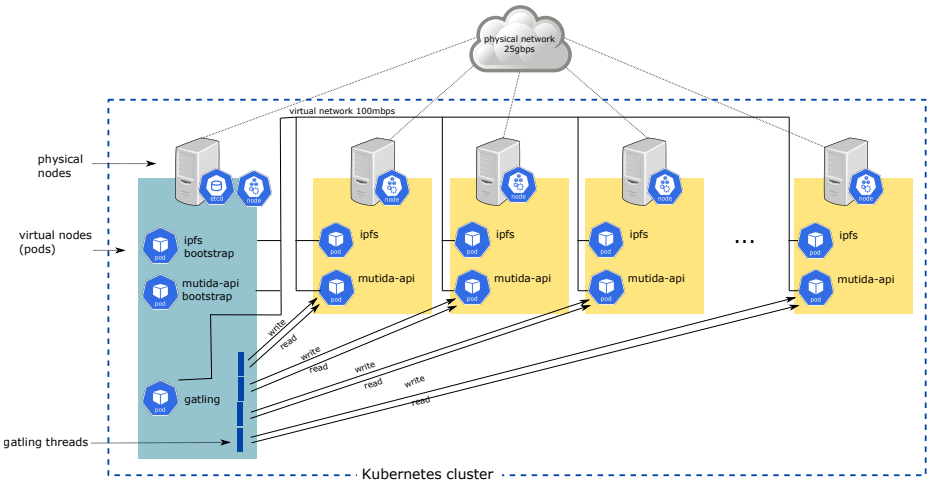


Fig. 9: Deployment of the solution in a Kubernetes cluster.

A single client is always hosted on the same physical node (co-located) using deployment constraints. The tests are carried out using the Gatling [22] software program, which sends requests to the aforementioned API. For all the propositions, the currently-implemented ACL mode is the replication mode, where a copy of the ACL's changes are sent in parallel to all nodes, and we wait for the responses in a synchronous way.

To evaluate the method, we propose three different scenarios: i) the first scenario entails evaluating the ACL without the impact of the data transfers and network exchanges, ii) the second scenario entails considering the network (i.e., data exchanged among different users) while keeping a uniform data access distribution, and iii) the third scenario is where we take a more realistic file distribution into consideration. In summary, we can describe the test scenarios as follows:

- **scenario i:** We encrypt data, create ACL entries, send files and access them locally, i.e., no data sharing with other users. The goal of this test is to evaluate the Mutida overhead in comparison with Shamir without the impact of the data sharing. In this scenario, 4 clients write n files; afterwards, the same 4 clients read them locally.
- **scenario ii:** We encrypt data, create ACL entries, send files, delegate rights to other users, and they access the files using the delegated values in the ACL. The goal of this test is to evaluate the total time impact that the Mutida approach would have in a complete scenario, including the data transaction. In this scenario, 4 clients write n files, and then 4 other clients read them.
- **scenario iii:** It is the same as scenario ii but follows a ZipF distribution [62] for files being accessed, where some files are more searched than others. The goals are the same as those from Scenario ii but rely on a more realistic file access distribution. In this scenario, n files are written on a single client and 3 different clients perform 100000 reads among these pieces of data

These scenarios are evaluated in the "Gros" cluster of the Grid'5000 platform [9], located in Nancy, France. The cluster is composed of 124 Intel Xeon Gold 5220 18 cores/CPU, SSD storage and is interconnected with a 2×25 Gbps network. To keep the scenarios closer to what would be a transfer occurring on the internet, we limit the network communication among different clients to 100 Mbps for Scenarios ii and iii. For each experiment, we allocate one dedicated machine per client and an extra one where Gatling, the certificate authority and the bootstrap for IPFS are hosted. In all cases, we consider that the managed files all have the same size of 1 MB. Each experiment was run 5 times to obtain consistency in the results.

As a base method for comparing the Mutida proposal, we rely on two different approaches. The first one is called classical encryption, and the second one is based on Shamir's secret sharing algorithm [53]. Details about how each of these approaches works and how they are implemented are described below.

5.1 Classical Cryptography

The first alternative that we explore is called classical cryptography. It consists of encrypting the file key with the public key of the user. We want to share the file with and include it in the public ACL (instead of the $ID1$ approach previously presented and used by Mutida). The comparison with this approach is restricted only to a first set of tests, where we compare the performance of each operation. We opt to use Shamir as a base comparison method because its additional functionalities (previously detailed) are closer to those in Mutida's method.

5.2 The Base Comparison Method - Shamir's Secret Sharing

As a base method for comparing Mutida in the previously presented scenarios, we rely on Shamir's secret sharing algorithm [53], which is one of the classical methods to secure a secret in a distributed way. The algorithm consists of splitting an arbitrary secret S into N parts called shares, and then distributing them among different peers. Among N parts, we can affirm that at least K is necessary to reconstruct the original secret S. We use this algorithm to split the decryption key of the files into multiple parts that are kept by different nodes. This way of sharing a secret enables us to ensure that a single malicious peer will not be able to reconstruct the secret (given that$K > 1$). The CodeHale[4] implementation of Shamir's operations is the one used in the experiments.

In Figure 10, the file is encrypted using a symmetric key. Then, the key is split into several parts that are spread among different ACL servers. The ACL servers also keep track of the users who are allowed to access the file. In Figure 11, we show the process to read a file, where the user has to contact different servers. Each server independently checks the user's permission before sending the key part. Then, when enough key parts are retrieved, the user can reconstruct the key and decrypt the file. This method is used in the next section to evaluate the performance of the proposal even if the security provided is different.

6 Results Evaluation

In this section, we start by presenting a brief comparison in a single node of each one of the operations that the protocol requires, compared to a classic public key encryption of the file key to highlight the protocol performance without any data exchange. Furthermore, we show a fully deployed solution for the three aforementioned scenarios and how to perform the Mutida method compared to a Shamir-based approach on those scenarios. The choice to use the Shamir approach relies on the similar functionalities that the method has, but we must keep in mind the increased level of security provided by Shamir during this comparison. In other words, if a single share of Shamir leaks, the whole key cannot be reconstructed. However, in the Mutida case, if one user with permissions on a specific file sees their private key stolen, the file accesses would be compromised.

[4] https://github.com/codahale/shamir

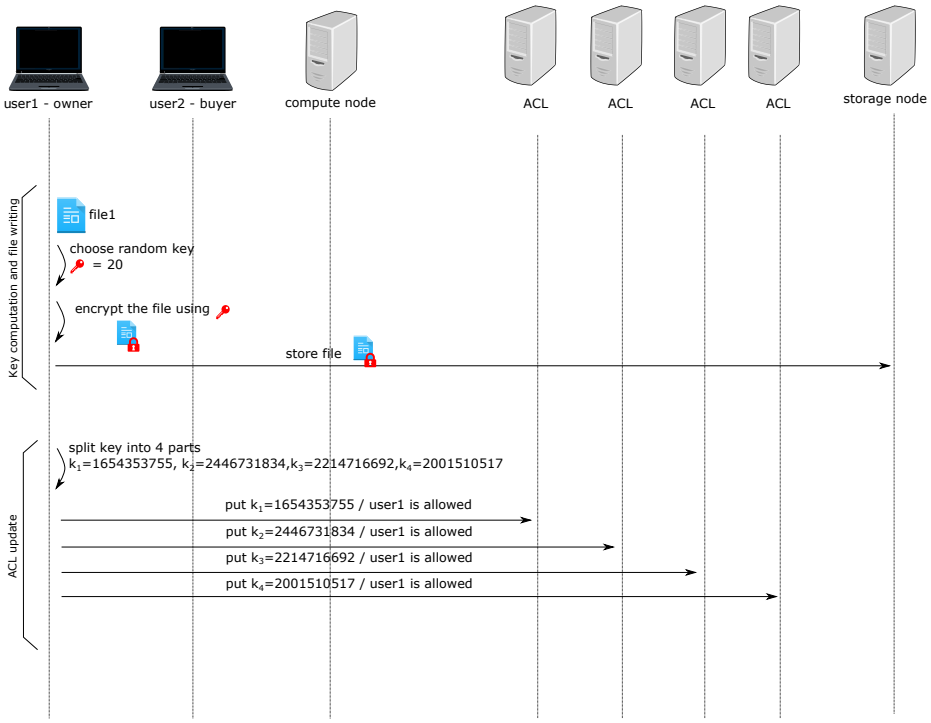


Fig. 10: Writing process using a protocol based on the Shamir algorithm.

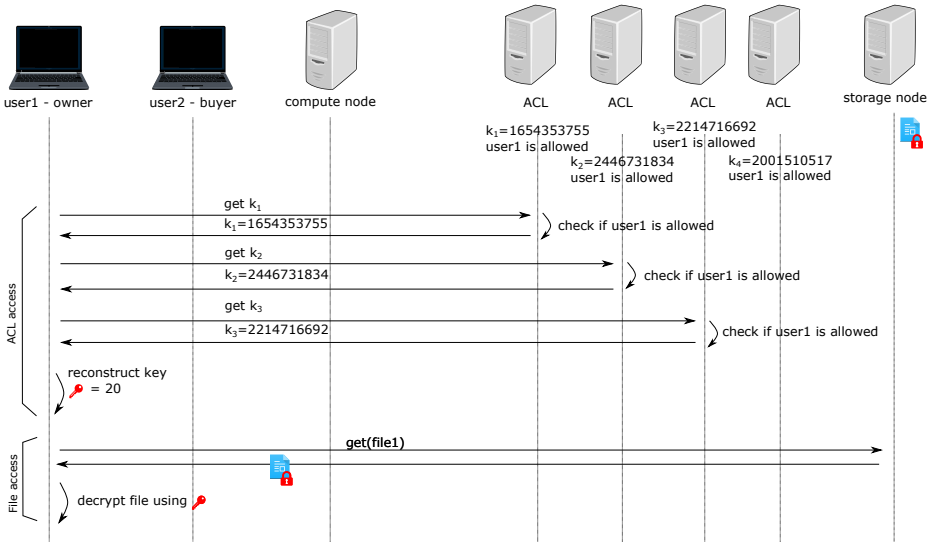


Fig. 11: Reading process using a protocol based on the Shamir algorithm.

6.1 Method calculation performance

Before evaluating the protocol in a distributed environment, we propose evaluating a single node, each operation necessary to accomplish the creation of a new entry, rights delegation and recalculating the encryption key. In Table 1, we present the average of 1000 runs for the Mutida approach, Shamir and what we consider to be classical cryptography.

We can observe that the standard deviation is often close to the average value (for instance, the time for "creating" in the classical approach with an average of 123.78 μs and a standard deviation of 751.83 μs). This imprecision of the measure is due to the process scheduler of the operating system and the timescale of the measurements. The total time corresponds to a basic scenario where a user creates a file, delegates the rights to a second user, and this second user computes the decryption key to perform a read.

Table 1 shows that the Mutida method is 57% faster than the classical approach to create a new entry in the ACL, and it is considerably faster to delegate and recalculate the file key. When comparing it to Shamir, which is used for the remaining experiments in this paper, we can see that the creation of a new ACL entry and the recalculation of the key are 80% and 97% faster, respectively, when compared with the Mutida method.

Operation	**Mutida**		**Classical**		**Shamir**	
	Time (μs)					
	Avg	Stdev	Avg	Stdev	Avg	Stdev
Create	**53.02**	97.28	123.78	751.83	272.06	199.23
Delegate	**10.05**	31.91	1770.82	382.11	N/A	N/A
Recalculate	**10.57**	44.19	1677.20	317.32	424.89	368.61
Total	73.64		3571.8		696.95	

Table 1: Time comparison of each operation for the Mutida, Shamir and classic cryptography

6.2 Scenario i: No data sharing

We begin by first evaluating how the two evaluated methods compare to one another with regard to the time spent writing, granting rights and reading a file stored in the local IPFS node. In Figure 12, we show that the difference in writing times between the two protocols is not substantial. This is because most of the time is spent in data transfer and i/o access in both approaches.

In Figures 13a and 13b, we present the time to write and read a new file using each proposition, without considering the file transfer operations and only considering the ones concerning the ACL and rights management operations. The operations that are considered in each case are described below.

When considering the protocol based on Shamir's method, the operations are (i) splitting the key, (ii) distribution of the parts on the different nodes, and (iii) encryption of the file. For reading, the operations are (i) retrieval of the key parts from the nodes, (ii) reconstruction of the key, and (iii) decryption of the file.

For Mutida's method, the operations considered for writing are (i) choosing a random number K for the ACL, (ii) computing the value of $SHA256(private\_key, filename)+K$, (iii) Storing the value K in the distributed ACL (on the different nodes), and (iv) Encryption of the file. For reading, the operations are (i) Computing the value of $SHA256(private\_key, filename) + K$ and (ii) Decryption of the file. We note that the ACL values and the key parts are spread in a synchronous way.

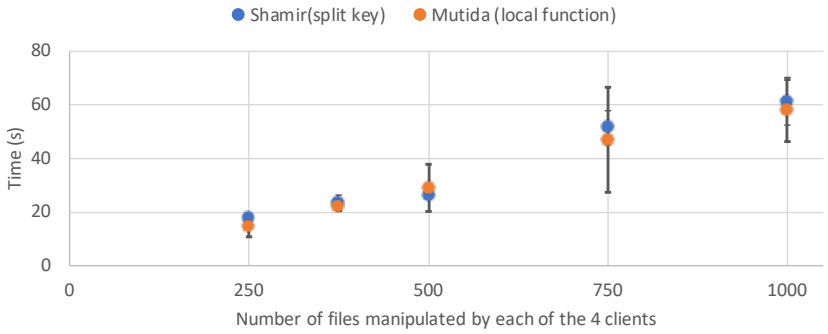
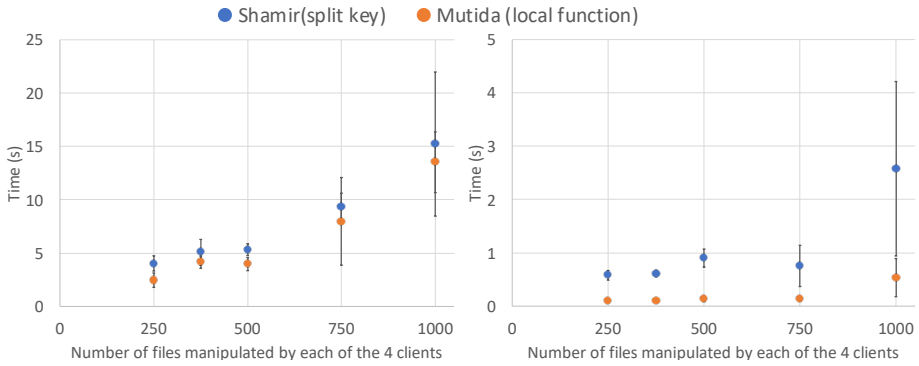


Fig. 12: Time for each client to encrypt {250,500,750,1000} files, to write them on their local server and to spread the key parts (Shamir) or the ACL value in the nodes.

Figures 13a and 13b show the access rights operations for the two solutions. It is between 5-10 seconds in writing and less than 1 second while reading. This is an important result because it means that when data are not shared, adding a protocol to manage access rights does not impact the performance.

In writing, our proposal has the same overhead as Shamir's solution because in the two approaches, ACL values need to be spread to all nodes. This network exchange is the operation that takes the most time and has more of an impact on the overhead. However, during the reading process, our approach has a lower overhead than the approach relying on a splitting key. This is because in our approach, the node only has to perform local operations (retrieving the ACL value and computing the ID ($ID(private\_key, filename)$ value), but in the Shamir approach, the node has to contact other nodes to retrieve the key parts.



(a) Write (b) Read

Fig. 13: Overhead of time due to the access right management.

6.3 Scenario ii: Sharing data with a single user

In this scenario, we evaluate the performance of the delegation of access rights and the performance when a user reads a file that it is not the owner of and the data are located in another peer.

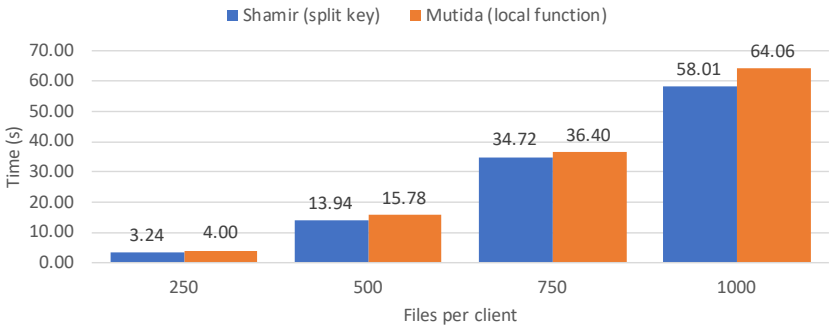


Fig. 14: Time to delegate the rights for the readers

Delegation of the access rights to the readers With regard to Shamir, the delegation consists of sending a request to the node that has previously written the file. The node propagates the request to all the nodes to ensure that they can record the fact that the reader is allowed to retrieve the key parts. In Mutida, the delegation is the protocol presented in Section 4. It consists of the reader computing the value ID $ID(private\_key, filename)$ and transmitting it to the node that wrote the file. The node computes the ACL value using its own private key and propagates this ACL value to all nodes that store it.

In the two propositions, we consider the time to realize the propagation to all the nodes. The main difference is that in our proposal, we have an extra exchange because the user has to compute a value that is transmitted to the file owner. Then, the file owner performs the propagation of the ACL value. This process is confirmed in Figure 14, which shows that the time to delegate access rights is more important with Mutida and is linear with the number of files.

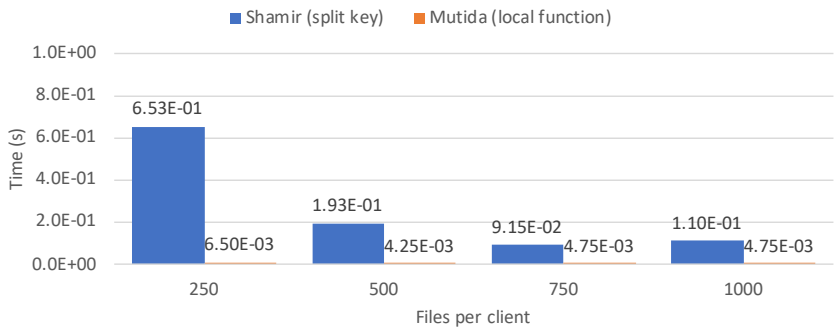


Fig. 15: Time to get the key to decrypt the file.

Reading of the files Figure 15 shows the amount of time that is required to obtain the key needed to decrypt the file using 4 clients. In Shamir, it corresponds to the time to obtain the key parts from other nodes and the time to execute the Shamir reconstruction algorithm. In Mutida, it is the time to compute $ID(private\_key, filename)$ and to add it to the local value of the ACL.

Despite the less time that rights delegation and key recalculation might take in comparison with data transfer and more costly network operations, this is still an important time and cost saving improvement.

6.4 Scenario iii: Real-world use case

In this scenario, each client writes 1000 files. Then, among all the files that have been written, each client generates 100,000 read requests using a zipf function ($skew = 0.5$). The zipf function represents the workload of a "real world" application [12, 62]. The delegation of access rights is performed during the first read, only once per couple (client, file). Then, only read operations are performed. The clients do not store the decryption key of the file recently read and reconstruct the key for each access.

Table 2 shows that in writing, Mutida is slightly slower than Shamir, but as shown by the standard deviation, this is not because Mutida is fundamentally slower; it is because of a lack of consistency in the test execution. Table 2 also shows that the approach using Mutida takes 8 times longer (1.23 s vs 0.16 s) to

delegate access rights than the approach using Shamir due to the calculation of the ID by the buyer and its transfer to the node of the owner.

However, this extra time is compensated by the different reads, since the time spent to recover the key parts needed by Shamir amount on average to 719.24 s compared to the 4.39 s spent by Mutida. Because some files are read several times but access delegation needs to be performed once, the total access time is shorter in the Mutida version than in the Shamir version when considering the full interaction of write, delegation and read. The average difference when considering the full scenario is more than 17 times slower (747.37 s vs 41.68 s) if we choose the Shamir approach.

		Shamir		Mutida	
		Average	**Stdev**	**Average**	**Stdev**
Write	ACL	12.03	5.77	24.60	7.98
	IPFS	15.54	6.62	10.50	3.89
	Total	27.57	8.82	35.10	10.96
Delegation		0.16	0.06	1.23	0.51
Read	ACL	**719.24**	141.68	**4.39**	9.15
	IPFS	0.40	0.14	0.96	0.17
	Total	719.64	141.62	5.35	9.47
Total		**747.37**		**41.68**	

Table 2: Time (in seconds) of write (1000 operations per client), delegation and read operations (100 000 operations per client).

6.5 Discussion of Results

The conducted experiments show that the amount of time to manage access rights and reconstruct the decryption keys is very small, leading to a small overhead. We start by highlighting that the Mutida method has a better performance when compared to classic cryptography, as well as when compared to Shamir's method. It requires less computational resources and has useful features, such as the impossibility of delegating access rights to a user who did not request it, as well as the possibility of giving temporary access to a computing node without adding a permanent record in the ACL.

Our first tests show that the most important overhead is not related to the computation but is due to the network traffic spreading ACL records between nodes. Further results show that Shamir and Mutida have a similar performance in writing a new entry because they execute similar operations, including generating a random key and propagating the ACL across all nodes. We also observe that Mutida has a considerably better performance in reading and reconstructing the file key. This is because the ACL is replicated on all the nodes, so Mutida is able to compute the decryption key without any network exchange, in contrast to Shamir, where all the key shares need to be recovered from the other peers.

Soliciting fewer of the other nodes means that it allows them fewer possibilities to act maliciously.

One of the main drawbacks observed in the method is related to rights revocation, but in this specific structure, it would be the same for all the current approaches. The difficulty of revoking the rights is related to the storage method for the ACL records and the use of immutable pieces of data rather than a weakness in the right management protocol.

Finally, we are also able to observe that the largest overhead of the whole process is related to the file transfer itself. Even if in this first moment we focus on the delegation mechanisms of Mutida, its performance compared to other methods from the literature, as well as the additional functionalities and the anonymization of the rights management, this overhead can make Mutida suitable for IoT environments that use files of only a few kilobytes [43].

7 Conclusion

In this paper, we introduce Mutida, a protocol to manage access rights in a distributed storage solution, which allows us to delegate these rights to other users and compute the nodes.

In comparison to standard approaches, Mutida differs in the following aspects: (i) Mutida has the ability to distrust the storage nodes to manage access rights; (ii) it has a low computational cost; (iii) it has low requirements for the users that only have to store their key to be able to decrypt all the pieces of data they can access; (iv) it has the ability to delegate access rights to users and compute nodes; and (v) it has the ability to remove the rights of compute nodes when the access rights of the user have been revoked. Additionally, we can use our approach coupled with a distributed P2P storage system allowing us to access the files even when the user is disconnected, without having to rely on them in a centralized server.

We begin by showing the time spent on each individual operation of the Mutida method compared with Shamir and the classical Public Key Cryptography, where Mutida takes almost half of the time in comparison with these other methods. After we present a quantitative analysis between the Mutida and Shamir approaches considering three different scenarios, the first scenario is without data sharing, where we can clearly see the overhead of Shamir, especially when we want to read a file. After the second and third scenarios where the data exchange takes place, we see, especially in scenario iii, the difference in the total time of the file exchange when comparing the two methods, where Shamir can be up to 17 times slower than Mutida.

As limitations of the protocol, because data are stored in an immutable and distributed way and despite the rights revocation, there is no way to ensure that there is no copy of the data stored and that a malicious user that once had access at some point did not store the keys; the revocation is not guaranteed. Finally, we aim to continue this work by performing a detailed security analysis of the

proposed mechanism and evaluating the long-term effects of this proposition in a production environment.

Acknowledgements Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

References

1. Adya, A., Bolosky, W.J., Castro, M., Cermak, G., Chaiken, R., Douceur, J.R., Howell, J., Lorch, J.R., Theimer, M., Wattenhofer, R.P.: Farsite: Federated, available, and reliable storage for an incompletely trusted environment. SIGOPS Oper. Syst. Rev. **36**(SI), 1–14 (Dec 2003). https://doi.org/10.1145/844128.844130, https://doi.org/10.1145/844128.844130
2. Ali, G., Ahmad, N., Cao, Y., Asif, M., Cruickshank, H., Ali, Q.E.: Blockchain based permission delegation and access control in internet of things (baci). Computers & Security **86**, 318 – 334 (2019). https://doi.org/https://doi.org/10.1016/j.cose.2019.06.010, http://www.sciencedirect.com/science/article/pii/S0167404819301208
3. Ali, M., Dhamotharan, R., Khan, E., Khan, S.U., Vasilakos, A.V., Li, K., Zomaya, A.Y.: Sedasc: Secure data sharing in clouds. IEEE Systems Journal **11**(2), 395–404 (2017). https://doi.org/10.1109/JSYST.2014.2379646
4. Andersen, M.P., Kumar, S., AbdelBaky, M., Fierro, G., Kolb, J., Kim, H.S., Culler, D.E., Popa, R.A.: Wave: A decentralized authorization framework with transitive delegation. In: Proceedings of the 28th USENIX Conference on Security Symposium. p. 1375–1392. SEC'19, USENIX Association, USA (2019)
5. Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., Muralidharan, S., Murthy, C., Nguyen, B., Sethi, M., Singh, G., Smith, K., Sorniotti, A., Stathakopoulou, C., Vukolić, M., Cocco, S.W., Yellick, J.: Hyperledger fabric: A distributed operating system for permissioned blockchains. In: Proceedings of the Thirteenth EuroSys Conference. EuroSys '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3190508.3190538, https://doi.org/10.1145/3190508.3190538
6. Angelis, S.D., Aniello, L., Baldoni, R., Lombardi, F., Margheri, A., Sassone, V.: Pbft vs proof-of-authority: applying the cap theorem to permissioned blockchain. In: Italian Conference on Cyber Security (06/02/18) (January 2018), https://eprints.soton.ac.uk/415083/
7. Aura, T.: Distributed Access-Rights Management with Delegation Certificates, pp. 211–235. Springer Berlin Heidelberg, Berlin, Heidelberg (1999)
8. Backes, M., Camenisch, J., Sommer, D.: Anonymous yet accountable access control. In: Proceedings of the 2005 ACM Workshop on Privacy in the Electronic Society. p. 40–46. WPES '05, Association for Computing Machinery, New York, NY, USA (2005). https://doi.org/10.1145/1102199.1102208, https://doi.org/10.1145/1102199.1102208
9. Balouek, D., Carpen Amarie, A., Charrier, G., Desprez, F., Jeannot, E., Jeanvoine, E., Lèbre, A., Margery, D., Niclausse, N., Nussbaum, L., Richard, O., Pérez,

C., Quesnel, F., Rohr, C., Sarzyniec, L.: Adding virtualization capabilities to the Grid'5000 testbed. In: Ivanov, I.I., van Sinderen, M., Leymann, F., Shan, T. (eds.) Cloud Computing and Services Science, Communications in Computer and Information Science, vol. 367, pp. 3–20. Springer International Publishing (2013). https://doi.org/10.1007/978-3-319-04519-1_1
10. Battah, A.A., Madine, M.M., Alzaabi, H., Yaqoob, I., Salah, K., Jayaraman, R.: Blockchain-based multi-party authorization for accessing ipfs encrypted data. IEEE Access **8**, 196813–196825 (2020). https://doi.org/10.1109/ACCESS.2020.3034260
11. Benet, J.: IPFS - Content Addressed, Versioned, P2P File System. Tech. rep., Protocol Labs, Inc. (2014), http://arxiv.org/abs/1407.3561
12. Breslau, L., Cao, P., Fan, L., Phillips, G., Shenker, S.: Web caching and zipf-like distributions: evidence and implications. In: IEEE INFOCOM '99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No.99CH36320). vol. 1, pp. 126–134 vol.1 (1999). https://doi.org/10.1109/INFCOM.1999.749260
13. Chaudhary, P., Gupta, R., Singh, A., Majumder, P.: Analysis and comparison of various fully homomorphic encryption techniques. In: 2019 International Conference on Computing, Power and Communication Technologies (GUCON). pp. 58–62 (2019)
14. Chen, J., Ma, H.: Efficient decentralized attribute-based access control for cloud storage with user revocation. In: 2014 IEEE International Conference on Communications (ICC). pp. 3782–3787 (2014). https://doi.org/10.1109/ICC.2014.6883910
15. Chow, S.S.M., Weng, J., Yang, Y., Deng, R.H.: Efficient unidirectional proxy re-encryption. In: Bernstein, D.J., Lange, T. (eds.) Progress in Cryptology – AFRICACRYPT 2010. pp. 316–332. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
16. Chuat, L., Abdou, A., Sasse, R., Sprenger, C., Basin, D., Perrig, A.: Sok: Delegation and revocation, the missing links in the web's chain of trust. In: 2020 IEEE European Symposium on Security and Privacy (EuroS P). pp. 624–638 (2020). https://doi.org/10.1109/EuroSP48549.2020.00046
17. Crampton, J., Khambhammettu, H.: Delegation in role-based access control. In: European Symposium on Research in Computer Security. pp. 174–191. Springer (2006)
18. Dang, Q.: Secure hash standard (2015-08-04 2015). https://doi.org/https://doi.org/10.6028/NIST.FIPS.180-4
19. Daswani, N., Garcia-Molina, H., Yang, B.: Open problems in data-sharing peer-to-peer systems. In: Proceedings of the 9th International Conference on Database Theory. p. 1–15. ICDT '03, Springer-Verlag, Berlin, Heidelberg (2003)
20. Fontaine, C., Galand, F.: A survey of homomorphic encryption for nonspecialists. EURASIP J. Inf. Secur. **2007**(1) (dec 2007)
21. Gasser, M., McDermott, E.: An architecture for practical delegation in a distributed system. In: 2012 IEEE Symposium on Security and Privacy. p. 20. IEEE Computer Society, Los Alamitos, CA, USA (may 1990). https://doi.org/10.1109/RISP.1990.63835, https://doi.ieeecomputersociety.org/10.1109/RISP.1990.63835
22. Gatling Corp: Gatling. https://gatling.io/ (2021), [Online; accessed 28-June-2021]
23. Gengler, B.: Content protection for recordable media (cprm). Computer Fraud & Security **2001**(2), 5–6 (2001).

https://doi.org/https://doi.org/10.1016/S1361-3723(01)02011-5, https://www.sciencedirect.com/science/article/pii/S1361372301020115
24. Hardt, D., et al.: The oauth 2.0 authorization framework (2012)
25. Heckmann, O., Bock, A., Mauthe, A., Steinmetz, R.: The edonkey file-sharing network. In: Dadam, P., Reichert, M. (eds.) Informatik 2004, Informatik verbindet, Band 2, Beiträge der 34. Jahrestagung der Gesellschaft für Informatik e.V. (GI). pp. 224–228. Gesellschaft für Informatik e.V., Bonn (2004)
26. Henningsen, S., Rust, S., Florian, M., Scheuermann, B.: Crawling the ipfs network. In: 2020 IFIP Networking Conference (Networking). pp. 679–680 (2020)
27. Hesselman, C., Moura, G.C., De Oliveira Schmidt, R., Toet, C.: Increasing dns security and stability through a control plane for top-level domain operators. IEEE Communications Magazine **55**(1), 197–203 (2017). https://doi.org/10.1109/MCOM.2017.1600521CM
28. Huu Tran, Hitchens, M., Varadharajan, V., Watters, P.: A trust based access control framework for p2p file-sharing systems. In: Proceedings of the 38th Annual Hawaii International Conference on System Sciences. pp. 302c–302c (2005)
29. Jawad, M., Alvarado, P.S., Valduriez, P.: Design of priserv, a privacy service for dhts. In: Proceedings of the 2008 International Workshop on Privacy and Anonymity in Information Society. p. 21–25. PAIS '08, Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1379287.1379293, https://doi.org/10.1145/1379287.1379293
30. Jin, H., Lotspiech, J.: Broadcast encryption for differently privileged. In: Gritzalis, D., Lopez, J. (eds.) Emerging Challenges for Security, Privacy and Trust. pp. 283–293. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
31. Katzarova, M., Simpson, A.: Delegation in a distributed healthcare context: A survey of current approaches. In: Katsikas, S.K., López, J., Backes, M., Gritzalis, S., Preneel, B. (eds.) Information Security. pp. 517–529. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
32. Kaushik, S., Gandhi, C.: Capability based outsourced data access control with assured file deletion and efficient revocation with trust factor in cloud computing. Int. J. Cloud Appl. Comput. **10**(1), 64–84 (jan 2020). https://doi.org/10.4018/IJCAC.2020010105, https://doi.org/10.4018/IJCAC.2020010105
33. Kieselmann, O., Kopal, N., Wacker, A.: A novel approach to data revocation on the internet. In: Garcia-Alfaro, J., Navarro-Arribas, G., Aldini, A., Martinelli, F., Suri, N. (eds.) Data Privacy Management, and Security Assurance. pp. 134–149. Springer International Publishing, Cham (2016)
34. Lasla, N., Younis, M., Znaidi, W., Ben Arbia, D.: Efficient distributed admission and revocation using blockchain for cooperative its. In: 2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS). pp. 1–5 (2018). https://doi.org/10.1109/NTMS.2018.8328734
35. Legout, A., Urvoy-Keller, G., Michiardi, P.: Understanding BitTorrent: An Experimental Perspective. Technical report, Inria (2005), https://hal.inria.fr/inria-00000156
36. Lesueur, F., Me, L., Tong, V.V.T.: An efficient distributed pki for structured p2p networks. In: 2009 IEEE Ninth International Conference on Peer-to-Peer Computing. pp. 1–10 (2009)
37. Liu, J., Li, X., Ye, L., Zhang, H., Du, X., Guizani, M.: Bpds: A blockchain based privacy-preserving data sharing for electronic medical records. In: 2018 IEEE Global Communications Conference (GLOBECOM). pp. 1–6 (2018)

38. Manousakis, K., Eswaran, S., Shur, D., Naik, G., Kantharaju, P., Regli, W., Adamson, B.: Torrent-based dissemination in infrastructure-less wireless networks. Journal of Cyber Security and Mobility **4**(1), 1–22 (2015)
39. Maymounkov, P., Mazières, D.: Kademlia: A peer-to-peer information system based on the xor metric. In: Revised Papers from the First International Workshop on Peer-to-Peer Systems. pp. 53–65. IPTPS '01, Springer-Verlag, London, UK, UK (2002), http://dl.acm.org/citation.cfm?id=646334.687801
40. Merkle, R.C.: Protocols for public key cryptosystems. In: 1980 IEEE Symposium on Security and Privacy. pp. 122–122 (1980). https://doi.org/10.1109/SP.1980.10006
41. Miller, S.P., Neuman, B.C., Schiller, J.I., Saltzer, J.H.: Kerberos authentication and authorization system. In: In Project Athena Technical Plan (1988)
42. Mockapetris, P.: Domain names - concepts and facilities. RFC 1034 (Nov 1987). https://doi.org/10.17487/RFC1034, https://rfc-editor.org/rfc/rfc1034.txt
43. Muralidharan, S., Ko, H.: An interplanetary file system (ipfs) based iot framework. In: 2019 IEEE International Conference on Consumer Electronics (ICCE). pp. 1–2 (2019). https://doi.org/10.1109/ICCE.2019.8662002
44. Naehrig, M., Lauter, K., Vaikuntanathan, V.: Can homomorphic encryption be practical? In: Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop. p. 113–124. CCSW '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/2046660.2046682, https://doi.org/10.1145/2046660.2046682
45. Nakatani, Y.: Structured allocation-based consistent hashing with improved balancing for cloud infrastructure. IEEE Transactions on Parallel and Distributed Systems **32**(9), 2248–2261 (2021). https://doi.org/10.1109/TPDS.2021.3058963
46. Politou, E., Alepis, E., Patsakis, C.: Forgetting personal data and revoking consent under the GDPR: Challenges and proposed solutions. Journal of Cybersecurity **4**(1) (03 2018). https://doi.org/10.1093/cybsec/tyy001, https://doi.org/10.1093/cybsec/tyy001, tyy001
47. Politou, E., Alepis, E., Patsakis, C., Casino, F., Alazab, M.: Delegated content erasure in ipfs. Future Generation Computer Systems **112**, 956–964 (2020). https://doi.org/https://doi.org/10.1016/j.future.2020.06.037, https://www.sciencedirect.com/science/article/pii/S0167739X19323003
48. Preneel, B.: Cryptographic hash functions. European Transactions on Telecommunications **5**(4), 431–448 (1994)
49. Protocol Labs: IPFS. https://ipfs.io/ (2021), [Online; accessed 28-June-2021]
50. Saleh, F.: Blockchain without Waste: Proof-of-Stake. The Review of Financial Studies **34**(3), 1156–1190 (07 2020). https://doi.org/10.1093/rfs/hhaa075, https://doi.org/10.1093/rfs/hhaa075
51. Sari, L., Sipos, M.: Filetribe: Blockchain-based secure file sharing on ipfs. In: European Wireless 2019; 25th European Wireless Conference. pp. 1–6 (2019)
52. Schnitzler, T., Dürmuth, M., Pöpper, C.: Towards contractual agreements for revocation of online data. In: Dhillon, G., Karlsson, F., Hedström, K., Zúquete, A. (eds.) ICT Systems Security and Privacy Protection. pp. 374–387. Springer International Publishing, Cham (2019)
53. Shamir, A.: How to share a secret. Commun. ACM **22**(11), 612–613 (Nov 1979). https://doi.org/10.1145/359168.359176, https://doi.org/10.1145/359168.359176
54. Steichen, M., Fiz, B., Norvill, R., Shbair, W., State, R.: Blockchain-based, decentralized access control for ipfs. In: 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData). pp. 1499–1506 (2018)

55. Stoica, I., Morris, R., Liben-Nowell, D., Karger, D., Kaashoek, M., Dabek, F., Balakrishnan, H.: Chord: a scalable peer-to-peer lookup protocol for internet applications. IEEE/ACM Transactions on Networking **11**(1), 17–32 (2003). https://doi.org/10.1109/TNET.2002.808407
56. The Linux Foundation: Kubernetes. https://kubernetes.io/ (2021), [Online; accessed 28-June-2021]
57. Urdaneta, G., Pierre, G., Steen, M.V.: A survey of dht security techniques. ACM Comput. Surv. **43**(2) (Feb 2011). https://doi.org/10.1145/1883612.1883615, https://doi.org/10.1145/1883612.1883615
58. Wang, S., Zhang, Y., Zhang, Y.: A blockchain-based framework for data sharing with fine-grained access control in decentralized storage systems. IEEE Access **6**, 38437–38450 (2018)
59. Wang, X., Sun, X., Sun, G., Luo, D.: Cst: P2p anonymous authentication system based on collaboration signature. In: 2010 5th International Conference on Future Information Technology. pp. 1–7 (2010). https://doi.org/10.1109/FUTURETECH.2010.5482740
60. Xu, R., Chen, Y., Blasch, E., Chen, G.: Blendcac: A smart contract enabled decentralized capability-based access control mechanism for the iot. Computers **7**(3) (2018). https://doi.org/10.3390/computers7030039, https://www.mdpi.com/2073-431X/7/3/39
61. Yang, W., Garg, S., Raza, A., Herbert, D., Kang, B.: Blockchain: Trends and future. In: Yoshida, K., Lee, M. (eds.) Knowledge Management and Acquisition for Intelligent Systems. pp. 201–210. Springer International Publishing, Cham (2018)
62. Yang, Y., Zhu, J.: Write skew and zipf distribution: Evidence and implications. ACM Trans. Storage **12**(4) (Jun 2016). https://doi.org/10.1145/2908557, https://doi.org/10.1145/2908557
63. Zhao, B., Huang, L., Stribling, J., Rhea, S., Joseph, A., Kubiatowicz, J.: Tapestry: a resilient global-scale overlay for service deployment. IEEE Journal on Selected Areas in Communications **22**(1), 41–53 (2004). https://doi.org/10.1109/JSAC.2003.818784